

Kinetis SDK v.2.1 API Reference Manual

NXP Semiconductors

Document Number: KSDK21K66FAPIRM
Rev. 0
Oct 2016



Contents

Chapter Introduction

Chapter Driver errors status

Chapter Architectural Overview

Chapter Trademarks

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

5.1	Overview	11
5.2	Typical use case	11
5.2.1	Polling Configuration	11
5.2.2	Interrupt Configuration	11
5.3	Data Structure Documentation	15
5.3.1	struct adc16_config_t	15
5.3.2	struct adc16_hardware_compare_config_t	16
5.3.3	struct adc16_channel_config_t	16
5.4	Macro Definition Documentation	17
5.4.1	FSL_ADC16_DRIVER_VERSION	17
5.5	Enumeration Type Documentation	17
5.5.1	_adc16_channel_status_flags	17
5.5.2	_adc16_status_flags	17
5.5.3	adc16_channel_mux_mode_t	17
5.5.4	adc16_clock_divider_t	18
5.5.5	adc16_resolution_t	18
5.5.6	adc16_clock_source_t	18
5.5.7	adc16_long_sample_mode_t	18
5.5.8	adc16_reference_voltage_source_t	19
5.5.9	adc16_hardware_average_mode_t	19
5.5.10	adc16_hardware_compare_mode_t	19
5.6	Function Documentation	19
5.6.1	ADC16_Init	19

Contents

Section Number	Title	Page Number
5.6.2	ADC16_Deinit	20
5.6.3	ADC16_GetDefaultConfig	20
5.6.4	ADC16_DoAutoCalibration	20
5.6.5	ADC16_SetOffsetValue	21
5.6.6	ADC16_EnableDMA	21
5.6.7	ADC16_EnableHardwareTrigger	21
5.6.8	ADC16_SetChannelMuxMode	22
5.6.9	ADC16_SetHardwareCompareConfig	22
5.6.10	ADC16_SetHardwareAverage	22
5.6.11	ADC16_GetStatusFlags	23
5.6.12	ADC16_ClearStatusFlags	23
5.6.13	ADC16_SetChannelConfig	23
5.6.14	ADC16_GetChannelConversionValue	25
5.6.15	ADC16_GetChannelStatusFlags	25

Chapter **CMP: Analog Comparator Driver**

6.1	Overview	27
6.2	Typical use case	27
6.2.1	Polling Configuration	27
6.2.2	Interrupt Configuration	27
6.3	Data Structure Documentation	30
6.3.1	struct cmp_config_t	30
6.3.2	struct cmp_filter_config_t	31
6.3.3	struct cmp_dac_config_t	31
6.4	Macro Definition Documentation	32
6.4.1	FSL_CMP_DRIVER_VERSION	32
6.5	Enumeration Type Documentation	32
6.5.1	_cmp_interrupt_enable	32
6.5.2	_cmp_status_flags	32
6.5.3	cmp_hysteresis_mode_t	32
6.5.4	cmp_reference_voltage_source_t	32
6.6	Function Documentation	33
6.6.1	CMP_Init	33
6.6.2	CMP_Deinit	33
6.6.3	CMP_Enable	33
6.6.4	CMP_GetDefaultConfig	34
6.6.5	CMP_SetInputChannels	34
6.6.6	CMP_EnableDMA	34
6.6.7	CMP_EnableWindowMode	35

Contents

Section Number	Title	Page Number
6.6.8	CMP_EnablePassThroughMode	35
6.6.9	CMP_SetFilterConfig	35
6.6.10	CMP_SetDACCConfig	35
6.6.11	CMP_EnableInterrupts	36
6.6.12	CMP_DisableInterrupts	36
6.6.13	CMP_GetStatusFlags	36
6.6.14	CMP_ClearStatusFlags	36
Chapter CMT: Carrier Modulator Transmitter Driver		
7.1	Overview	39
7.2	Clock formulas	39
7.3	Typical use case	39
7.4	Data Structure Documentation	42
7.4.1	struct cmt_modulate_config_t	42
7.4.2	struct cmt_config_t	43
7.5	Macro Definition Documentation	43
7.5.1	FSL_CMT_DRIVER_VERSION	43
7.6	Enumeration Type Documentation	43
7.6.1	cmt_mode_t	43
7.6.2	cmt_primary_clkdiv_t	44
7.6.3	cmt_second_clkdiv_t	44
7.6.4	cmt_infrared_output_polarity_t	45
7.6.5	cmt_infrared_output_state_t	45
7.6.6	_cmt_interrupt_enable	45
7.7	Function Documentation	45
7.7.1	CMT_GetDefaultConfig	45
7.7.2	CMT_Init	45
7.7.3	CMT_Deinit	46
7.7.4	CMT_SetMode	46
7.7.5	CMT_GetMode	46
7.7.6	CMT_GetCMTFrequency	47
7.7.7	CMT_SetCarrirGenerateCountOne	48
7.7.8	CMT_SetCarrirGenerateCountTwo	48
7.7.9	CMT_SetModulateMarkSpace	49
7.7.10	CMT_EnableExtendedSpace	49
7.7.11	CMT_SetIroState	50
7.7.12	CMT_EnableInterrupts	51
7.7.13	CMT_DisableInterrupts	51
7.7.14	CMT_GetStatusFlags	51

Contents

Section Number	Title	Page Number
Chapter	CRC: Cyclic Redundancy Check Driver	
8.1	Overview	53
8.2	CRC Driver Initialization and Configuration	53
8.3	CRC Write Data	53
8.4	CRC Get Checksum	53
8.5	Comments about API usage in RTOS	54
8.6	Comments about API usage in interrupt handler	54
8.7	CRC Driver Examples	54
8.7.1	Simple examples	54
8.7.2	Advanced examples	55
8.8	Data Structure Documentation	58
8.8.1	struct crc_config_t	58
8.9	Macro Definition Documentation	58
8.9.1	FSL_CRC_DRIVER_VERSION	58
8.9.2	CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT	59
8.10	Enumeration Type Documentation	59
8.10.1	crc_bits_t	59
8.10.2	crc_result_t	59
8.11	Function Documentation	59
8.11.1	CRC_Init	59
8.11.2	CRC_Deinit	59
8.11.3	CRC_GetDefaultConfig	60
8.11.4	CRC_WriteData	60
8.11.5	CRC_Get32bitResult	60
8.11.6	CRC_Get16bitResult	61
Chapter	DAC: Digital-to-Analog Converter Driver	
9.1	Overview	63
9.2	Typical use case	63
9.2.1	Working as a basic DAC without the hardware buffer feature	63
9.2.2	Working with the hardware buffer	63
9.3	Data Structure Documentation	66
9.3.1	struct dac_config_t	66

Contents

Section Number	Title	Page Number
9.3.2	<code>struct dac_buffer_config_t</code>	66
9.4	Macro Definition Documentation	67
9.4.1	<code>FSL_DAC_DRIVER_VERSION</code>	67
9.5	Enumeration Type Documentation	67
9.5.1	<code>_dac_buffer_status_flags</code>	67
9.5.2	<code>_dac_buffer_interrupt_enable</code>	67
9.5.3	<code>dac_reference_voltage_source_t</code>	68
9.5.4	<code>dac_buffer_trigger_mode_t</code>	68
9.5.5	<code>dac_buffer_watermark_t</code>	68
9.5.6	<code>dac_buffer_work_mode_t</code>	68
9.6	Function Documentation	68
9.6.1	<code>DAC_Init</code>	68
9.6.2	<code>DAC_Deinit</code>	69
9.6.3	<code>DAC_GetDefaultConfig</code>	69
9.6.4	<code>DAC_Enable</code>	69
9.6.5	<code>DAC_EnableBuffer</code>	70
9.6.6	<code>DAC_SetBufferConfig</code>	70
9.6.7	<code>DAC_GetDefaultBufferConfig</code>	70
9.6.8	<code>DAC_EnableBufferDMA</code>	70
9.6.9	<code>DAC_SetBufferValue</code>	71
9.6.10	<code>DAC_DoSoftwareTriggerBuffer</code>	71
9.6.11	<code>DAC_GetBufferReadPointer</code>	71
9.6.12	<code>DAC_SetBufferReadPointer</code>	72
9.6.13	<code>DAC_EnableBufferInterrupts</code>	72
9.6.14	<code>DAC_DisableBufferInterrupts</code>	72
9.6.15	<code>DAC_GetBufferStatusFlags</code>	72
9.6.16	<code>DAC_ClearBufferStatusFlags</code>	72

Chapter DMAMUX: Direct Memory Access Multiplexer Driver

10.1	Overview	75
10.2	Typical use case	75
10.2.1	<code>DMAMUX Operation</code>	75
10.3	Macro Definition Documentation	75
10.3.1	<code>FSL_DMAMUX_DRIVER_VERSION</code>	75
10.4	Function Documentation	76
10.4.1	<code>DMAMUX_Init</code>	76
10.4.2	<code>DMAMUX_Deinit</code>	77
10.4.3	<code>DMAMUX_EnableChannel</code>	77
10.4.4	<code>DMAMUX_DisableChannel</code>	77

Contents

Section Number	Title	Page Number
10.4.5	DMAMUX_SetSource	78
10.4.6	DMAMUX_EnablePeriodTrigger	78
10.4.7	DMAMUX_DisablePeriodTrigger	78
Chapter	DSPI: Serial Peripheral Interface Driver	
11.1	Overview	79
11.2	DSPI Driver	80
11.2.1	Overview	80
11.2.2	Typical use case	80
11.2.3	Data Structure Documentation	87
11.2.4	Macro Definition Documentation	94
11.2.5	Typedef Documentation	95
11.2.6	Enumeration Type Documentation	96
11.2.7	Function Documentation	100
11.3	DSPI DMA Driver	118
11.3.1	Overview	118
11.3.2	Data Structure Documentation	119
11.3.3	Typedef Documentation	122
11.3.4	Function Documentation	123
11.4	DSPI eDMA Driver	128
11.4.1	Overview	128
11.4.2	Data Structure Documentation	129
11.4.3	Typedef Documentation	132
11.4.4	Function Documentation	133
11.5	DSPI FreeRTOS Driver	138
11.5.1	Overview	138
11.5.2	Function Documentation	138
11.6	DSPI μCOS/II Driver	140
11.6.1	Overview	140
11.6.2	Function Documentation	140
11.7	DSPI μCOS/III Driver	142
11.7.1	Overview	142
11.7.2	Function Documentation	142
Chapter	eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	
12.1	Overview	145
12.2	Typical use case	145

Contents

Section Number	Title	Page Number
12.2.1	eDMA Operation	145
12.3	Data Structure Documentation	151
12.3.1	struct edma_config_t	151
12.3.2	struct edma_transfer_config_t	151
12.3.3	struct edma_channel_Preemption_config_t	152
12.3.4	struct edma_minor_offset_config_t	153
12.3.5	struct edma_tcd_t	153
12.3.6	struct edma_handle_t	154
12.4	Macro Definition Documentation	155
12.4.1	FSL_EDMA_DRIVER_VERSION	155
12.5	Typedef Documentation	155
12.5.1	edma_callback	155
12.6	Enumeration Type Documentation	155
12.6.1	edma_transfer_size_t	155
12.6.2	edma_modulo_t	155
12.6.3	edma_bandwidth_t	156
12.6.4	edma_channel_link_type_t	156
12.6.5	_edma_channel_status_flags	156
12.6.6	_edma_error_status_flags	157
12.6.7	edma_interrupt_enable_t	157
12.6.8	edma_transfer_type_t	157
12.6.9	_edma_transfer_status	158
12.7	Function Documentation	158
12.7.1	EDMA_Init	158
12.7.2	EDMA_Deinit	158
12.7.3	EDMA_GetDefaultConfig	158
12.7.4	EDMA_ResetChannel	159
12.7.5	EDMA_SetTransferConfig	159
12.7.6	EDMA_SetMinorOffsetConfig	160
12.7.7	EDMA_SetChannelPreemptionConfig	160
12.7.8	EDMA_SetChannelLink	160
12.7.9	EDMA_SetBandWidth	162
12.7.10	EDMA_SetModulo	163
12.7.11	EDMA_EnableAsyncRequest	163
12.7.12	EDMA_EnableAutoStopRequest	163
12.7.13	EDMA_EnableChannelInterrupts	164
12.7.14	EDMA_DisableChannelInterrupts	165
12.7.15	EDMA_TcdReset	165
12.7.16	EDMA_TcdSetTransferConfig	165
12.7.17	EDMA_TcdSetMinorOffsetConfig	166

Contents

Section Number	Title	Page Number
12.7.18	EDMA_TcdSetChannelLink	166
12.7.19	EDMA_TcdSetBandWidth	167
12.7.20	EDMA_TcdSetModulo	167
12.7.21	EDMA_TcdEnableAutoStopRequest	168
12.7.22	EDMA_TcdEnableInterrupts	169
12.7.23	EDMA_TcdDisableInterrupts	169
12.7.24	EDMA_EnableChannelRequest	169
12.7.25	EDMA_DisableChannelRequest	169
12.7.26	EDMA_TriggerChannelStart	170
12.7.27	EDMA_GetRemainingMajorLoopCount	170
12.7.28	EDMA_GetErrorStatusFlags	171
12.7.29	EDMA_GetChannelStatusFlags	171
12.7.30	EDMA_ClearChannelStatusFlags	171
12.7.31	EDMA_CreateHandle	172
12.7.32	EDMA_InstallTCMDMemory	173
12.7.33	EDMA_SetCallback	173
12.7.34	EDMA_PreparesTransfer	173
12.7.35	EDMA_SubmitTransfer	174
12.7.36	EDMA_StartTransfer	175
12.7.37	EDMA_StopTransfer	175
12.7.38	EDMA_AbortTransfer	175
12.7.39	EDMA_HandleIRQ	175

Chapter ENET: Ethernet MAC Driver

13.1	Overview	177
13.2	Typical use case	177
13.2.1	ENET Initialization, receive, and transmit operations	177
13.3	Data Structure Documentation	185
13.3.1	struct enet_rx_bd_struct_t	185
13.3.2	struct enet_tx_bd_struct_t	185
13.3.3	struct enet_data_error_stats_t	186
13.3.4	struct enet_buffer_config_t	186
13.3.5	struct enet_config_t	187
13.3.6	struct _enet_handle	189
13.4	Macro Definition Documentation	190
13.4.1	FSL_ENET_DRIVER_VERSION	190
13.4.2	ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK	192
13.4.3	ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK	192
13.4.4	ENET_BUFFDESCRIPTOR_RX_WRAP_MASK	192
13.4.5	ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask	192
13.4.6	ENET_BUFFDESCRIPTOR_RX_LAST_MASK	192

Contents

Section Number	Title	Page Number
13.4.7	ENET_BUFFDESCRIPTOR_RX_MISS_MASK	192
13.4.8	ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK	192
13.4.9	ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK	192
13.4.10	ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK	192
13.4.11	ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK	192
13.4.12	ENET_BUFFDESCRIPTOR_RX_CRC_MASK	192
13.4.13	ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK	192
13.4.14	ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK	192
13.4.15	ENET_BUFFDESCRIPTOR_TX_READY_MASK	192
13.4.16	ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK	192
13.4.17	ENET_BUFFDESCRIPTOR_TX_WRAP_MASK	192
13.4.18	ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK	192
13.4.19	ENET_BUFFDESCRIPTOR_TX_LAST_MASK	192
13.4.20	ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK	192
13.4.21	ENET_BUFFDESCRIPTOR_RX_ERR_MASK	192
13.4.22	ENET_FRAME_MAX_FRAMELEN	193
13.4.23	ENET_FRAME_MAX_VALNFRAMELEN	193
13.4.24	ENET_FIFO_MIN_RX_FULL	193
13.4.25	ENET_RX_MIN_BUFSIZE	193
13.4.26	ENET_BUFF_ALIGNMENT	193
13.4.27	ENET_PHY_MAXADDRESS	193
13.5	Typedef Documentation	193
13.5.1	enet_callback_t	193
13.6	Enumeration Type Documentation	193
13.6.1	_enet_status	193
13.6.2	enet_mii_mode_t	193
13.6.3	enet_mii_speed_t	194
13.6.4	enet_mii_duplex_t	194
13.6.5	enet_mii_write_t	194
13.6.6	enet_mii_read_t	194
13.6.7	enet_special_control_flag_t	194
13.6.8	enet_interrupt_enable_t	195
13.6.9	enet_event_t	195
13.6.10	enet_tx_accelerator_t	196
13.6.11	enet_rx_accelerator_t	196
13.7	Function Documentation	196
13.7.1	ENET_GetDefaultConfig	196
13.7.2	ENET_Init	196
13.7.3	ENET_Deinit	197
13.7.4	ENET_Reset	197
13.7.5	ENET_SetMII	198
13.7.6	ENET_SetSMI	198

Contents

Section Number	Title	Page Number
13.7.7	ENET_GetSMI	198
13.7.8	ENET_ReadSMIData	199
13.7.9	ENET_StartSMIRead	200
13.7.10	ENET_StartSMIWrite	200
13.7.11	ENET_SetMacAddr	200
13.7.12	ENET_GetMacAddr	201
13.7.13	ENET_AddMulticastGroup	201
13.7.14	ENET_LeaveMulticastGroup	201
13.7.15	ENET_ActiveRead	201
13.7.16	ENET_EnableSleepMode	202
13.7.17	ENET_GetAccelFunction	202
13.7.18	ENET_EnableInterrupts	202
13.7.19	ENET_DisableInterrupts	204
13.7.20	ENET_GetInterruptStatus	204
13.7.21	ENET_ClearInterruptStatus	204
13.7.22	ENET_SetCallback	205
13.7.23	ENET_GetRxErrBeforeReadFrame	205
13.7.24	ENET_GetRxFrameSize	206
13.7.25	ENET_ReadFrame	206
13.7.26	ENET_SendFrame	207
13.7.27	ENET_TransmitIRQHandler	208
13.7.28	ENET_ReceiveIRQHandler	208
13.7.29	ENET_ErrorIRQHandler	208

Chapter **EWM: External Watchdog Monitor Driver**

14.1	Overview	209
14.2	Typical use case	209
14.3	Data Structure Documentation	210
14.3.1	struct ewm_config_t	210
14.4	Macro Definition Documentation	210
14.4.1	FSL_EWM_DRIVER_VERSION	210
14.5	Enumeration Type Documentation	210
14.5.1	_ewm_interrupt_enable_t	210
14.5.2	_ewm_status_flags_t	210
14.6	Function Documentation	211
14.6.1	EWM_Init	211
14.6.2	EWM_Deinit	211
14.6.3	EWM_GetDefaultConfig	211
14.6.4	EWM_EnableInterrupts	212

Contents

Section Number	Title	Page Number
14.6.5	EWM_DisableInterrupts	212
14.6.6	EWM_GetStatusFlags	212
14.6.7	EWM_Refresh	213

Chapter C90TFS Flash Driver

15.1	Overview	215
15.2	Data Structure Documentation	223
15.2.1	struct flash_execute_in_ram_function_config_t	223
15.2.2	struct flash_swap_state_config_t	223
15.2.3	struct flash_swap_ifr_field_config_t	223
15.2.4	union flash_swap_ifr_field_data_t	224
15.2.5	union pflash_protection_status_low_t	224
15.2.6	struct pflash_protection_status_t	225
15.2.7	struct flash_prefetch_speculation_status_t	225
15.2.8	struct flash_protection_config_t	225
15.2.9	struct flash_access_config_t	226
15.2.10	struct flash_operation_config_t	226
15.2.11	struct flash_config_t	227
15.3	Macro Definition Documentation	228
15.3.1	MAKE_VERSION	228
15.3.2	FSL_FLASH_DRIVER_VERSION	228
15.3.3	FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT	228
15.3.4	FLASH_DRIVER_IS_FLASH_RESIDENT	228
15.3.5	FLASH_DRIVER_IS_EXPORTED	229
15.3.6	kStatusGroupGeneric	229
15.3.7	MAKE_STATUS	229
15.3.8	FOUR_CHAR_CODE	229
15.4	Enumeration Type Documentation	229
15.4.1	_flash_driver_version_constants	229
15.4.2	_flash_status	229
15.4.3	_flash_driver_api_keys	230
15.4.4	flash_margin_value_t	230
15.4.5	flash_security_state_t	230
15.4.6	flash_protection_state_t	231
15.4.7	flash_execute_only_access_state_t	231
15.4.8	flash_property_tag_t	231
15.4.9	_flash_execute_in_ram_function_constants	232
15.4.10	flash_read_resource_option_t	232
15.4.11	_flash_read_resource_range	232
15.4.12	flash_flexram_function_option_t	232
15.4.13	flash_swap_function_option_t	233

Contents

Section Number	Title	Page Number
15.4.14	<code>flash_swap_control_option_t</code>	233
15.4.15	<code>flash_swap_state_t</code>	233
15.4.16	<code>flash_swap_block_status_t</code>	233
15.4.17	<code>flash_partition_flexram_load_option_t</code>	234
15.4.18	<code>flash_memory_index_t</code>	234
15.5	Function Documentation	234
15.5.1	<code>FLASH_Init</code>	234
15.5.2	<code>FLASH_SetCallback</code>	234
15.5.3	<code>FLASH_PrepareExecuteInRamFunctions</code>	235
15.5.4	<code>FLASH_EraseAll</code>	235
15.5.5	<code>FLASH_Erase</code>	236
15.5.6	<code>FLASH_EraseAllExecuteOnlySegments</code>	237
15.5.7	<code>FLASH_Program</code>	239
15.5.8	<code>FLASH_ProgramOnce</code>	240
15.5.9	<code>FLASH_ProgramSection</code>	240
15.5.10	<code>FLASH_EepromWrite</code>	241
15.5.11	<code>FLASH_ReadResource</code>	242
15.5.12	<code>FLASH_ReadOnce</code>	243
15.5.13	<code>FLASH_GetSecurityState</code>	244
15.5.14	<code>FLASH_SecurityBypass</code>	244
15.5.15	<code>FLASH_VerifyEraseAll</code>	245
15.5.16	<code>FLASH_VerifyErase</code>	246
15.5.17	<code>FLASH_VerifyProgram</code>	247
15.5.18	<code>FLASH_VerifyEraseAllExecuteOnlySegments</code>	248
15.5.19	<code>FLASH_IsProtected</code>	249
15.5.20	<code>FLASH_IsExecuteOnly</code>	249
15.5.21	<code>FLASHGetProperty</code>	250
15.5.22	<code>FLASH_SetFlexramFunction</code>	250
15.5.23	<code>FLASH_SwapControl</code>	251
15.5.24	<code>FLASH_Swap</code>	252
15.5.25	<code>FLASH_ProgramPartition</code>	253
15.5.26	<code>FLASH_PflashSetProtection</code>	254
15.5.27	<code>FLASH_PflashGetProtection</code>	254
15.5.28	<code>FLASH_DflashSetProtection</code>	255
15.5.29	<code>FLASH_DflashGetProtection</code>	255
15.5.30	<code>FLASH_EepromSetProtection</code>	256
15.5.31	<code>FLASH_EepromGetProtection</code>	257

Chapter **FlexBus: External Bus Interface Driver**

16.1	Overview	259
16.2	FlexBus functional operation	259

Contents

Section Number	Title	Page Number
16.3	Typical use case and example	259
16.4	Data Structure Documentation	261
16.4.1	struct flexbus_config_t	261
16.5	Macro Definition Documentation	262
16.5.1	FSL_FLEXBUS_DRIVER_VERSION	262
16.6	Enumeration Type Documentation	262
16.6.1	flexbus_port_size_t	262
16.6.2	flexbus_write_address_hold_t	262
16.6.3	flexbus_read_address_hold_t	263
16.6.4	flexbus_address_setup_t	263
16.6.5	flexbus_bytelane_shift_t	263
16.6.6	flexbus_multiplex_group1_t	263
16.6.7	flexbus_multiplex_group2_t	264
16.6.8	flexbus_multiplex_group3_t	264
16.6.9	flexbus_multiplex_group4_t	264
16.6.10	flexbus_multiplex_group5_t	264
16.7	Function Documentation	264
16.7.1	FLEXBUS_Init	264
16.7.2	FLEXBUS_Deinit	265
16.7.3	FLEXBUS_GetDefaultConfig	265
Chapter FlexCAN: Flex Controller Area Network Driver		
17.1	Overview	267
17.2	FlexCAN Driver	268
17.2.1	Overview	268
17.2.2	Typical use case	268
17.2.3	Data Structure Documentation	276
17.2.4	Macro Definition Documentation	280
17.2.5	Typedef Documentation	285
17.2.6	Enumeration Type Documentation	285
17.2.7	Function Documentation	288
17.3	FlexCAN eDMA Driver	302
17.3.1	Overview	302
17.3.2	Data Structure Documentation	302
17.3.3	Typedef Documentation	303
17.3.4	Function Documentation	303

Contents

Section Number	Title	Page Number
Chapter	FTM: FlexTimer Driver	
18.1	Overview	307
18.2	Function groups	307
18.2.1	Initialization and deinitialization	307
18.2.2	PWM Operations	307
18.2.3	Input capture operations	307
18.2.4	Output compare operations	308
18.2.5	Quad decode	308
18.2.6	Fault operation	308
18.3	Register Update	308
18.4	Typical use case	309
18.4.1	PWM output	309
18.5	Data Structure Documentation	315
18.5.1	struct ftm_chnl_pwm_signal_param_t	315
18.5.2	struct ftm_dual_edge_capture_param_t	316
18.5.3	struct ftm_phase_params_t	316
18.5.4	struct ftm_fault_param_t	317
18.5.5	struct ftm_config_t	317
18.6	Enumeration Type Documentation	318
18.6.1	ftm_chnl_t	318
18.6.2	ftm_fault_input_t	318
18.6.3	ftm_pwm_mode_t	319
18.6.4	ftm_pwm_level_select_t	319
18.6.5	ftm_output_compare_mode_t	319
18.6.6	ftm_input_capture_edge_t	319
18.6.7	ftm_dual_edge_capture_mode_t	319
18.6.8	ftm_quad_decode_mode_t	320
18.6.9	ftm_phase_polarity_t	320
18.6.10	ftm_deadtime_prescale_t	320
18.6.11	ftm_clock_source_t	320
18.6.12	ftm_clock_prescale_t	320
18.6.13	ftm_bdm_mode_t	321
18.6.14	ftm_fault_mode_t	321
18.6.15	ftm_external_trigger_t	321
18.6.16	ftm_pwm_sync_method_t	322
18.6.17	ftm_reload_point_t	322
18.6.18	ftm_interrupt_enable_t	322
18.6.19	ftm_status_flags_t	323
18.6.20	_ftm_quad_decoder_flags	323

Contents

Section Number	Title	Page Number
18.7	Function Documentation	323
18.7.1	FTM_Init	323
18.7.2	FTM_Deinit	324
18.7.3	FTM_GetDefaultConfig	324
18.7.4	FTM_SetupPwm	324
18.7.5	FTM_UpdatePwmDutycycle	325
18.7.6	FTM_UpdateChnlEdgeLevelSelect	325
18.7.7	FTM_SetupInputCapture	326
18.7.8	FTM_SetupOutputCompare	326
18.7.9	FTM_SetupDualEdgeCapture	327
18.7.10	FTM_SetupFault	327
18.7.11	FTM_EnableInterrupts	327
18.7.12	FTM_DisableInterrupts	328
18.7.13	FTM_GetEnabledInterrupts	328
18.7.14	FTM_GetStatusFlags	328
18.7.15	FTM_ClearStatusFlags	328
18.7.16	FTM_StartTimer	329
18.7.17	FTM_StopTimer	329
18.7.18	FTM_SetSoftwareCtrlEnable	329
18.7.19	FTM_SetSoftwareCtrlVal	329
18.7.20	FTM_SetGlobalTimeBaseOutputEnable	330
18.7.21	FTM_SetOutputMask	330
18.7.22	FTM_SetFaultControlEnable	330
18.7.23	FTM_SetDeadTimeEnable	331
18.7.24	FTM_SetComplementaryEnable	331
18.7.25	FTM_SetInvertEnable	331
18.7.26	FTM_SetupQuadDecode	332
18.7.27	FTM_GetQuadDecoderFlags	332
18.7.28	FTM_SetQuadDecoderModuloValue	332
18.7.29	FTM_GetQuadDecoderCounterValue	333
18.7.30	FTM_ClearQuadDecoderCounterValue	333
18.7.31	FTM_SetSoftwareTrigger	333
18.7.32	FTM_SetWriteProtection	333

Chapter 19 GPIO: General-Purpose Input/Output Driver

19.1	Overview	335
19.2	Data Structure Documentation	335
19.2.1	struct gpio_pin_config_t	335
19.3	Macro Definition Documentation	336
19.3.1	FSL_GPIO_DRIVER_VERSION	336
19.4	Enumeration Type Documentation	336

Contents

Section Number	Title	Page Number
19.4.1	gpio_pin_direction_t	336
19.5	GPIO Driver	337
19.5.1	Overview	337
19.5.2	Typical use case	337
19.5.3	Function Documentation	338
19.6	FGPIO Driver	341
19.6.1	Typical use case	341
Chapter	I2C: Inter-Integrated Circuit Driver	
20.1	Overview	343
20.2	I2C Driver	344
20.2.1	Overview	344
20.2.2	Typical use case	344
20.2.3	Data Structure Documentation	351
20.2.4	Macro Definition Documentation	356
20.2.5	Typedef Documentation	356
20.2.6	Enumeration Type Documentation	356
20.2.7	Function Documentation	358
20.3	I2C eDMA Driver	373
20.3.1	Overview	373
20.3.2	Data Structure Documentation	373
20.3.3	Typedef Documentation	374
20.3.4	Function Documentation	374
20.4	I2C DMA Driver	377
20.4.1	Overview	377
20.4.2	Data Structure Documentation	377
20.4.3	Typedef Documentation	378
20.4.4	Function Documentation	378
20.5	I2C FreeRTOS Driver	380
20.5.1	Overview	380
20.5.2	Function Documentation	380
20.6	I2C µCOS/II Driver	382
20.6.1	Overview	382
20.6.2	Function Documentation	382
20.7	I2C µCOS/III Driver	384
20.7.1	Overview	384
20.7.2	Function Documentation	384

Contents

Section Number	Title	Page Number
Chapter	LLWU: Low-Leakage Wakeup Unit Driver	
21.1	Overview	387
21.2	External wakeup pins configurations	387
21.3	Internal wakeup modules configurations	387
21.4	Digital pin filter for external wakeup pin configurations	387
21.5	Data Structure Documentation	388
21.5.1	struct llwu_external_pin_filter_mode_t	388
21.6	Macro Definition Documentation	388
21.6.1	FSL_LLWU_DRIVER_VERSION	388
21.7	Enumeration Type Documentation	388
21.7.1	llwu_external_pin_mode_t	388
21.7.2	llwu_pin_filter_mode_t	389
21.8	Function Documentation	389
21.8.1	LLWU_SetExternalWakeupsPinMode	389
21.8.2	LLWU_GetExternalWakeupsPinFlag	389
21.8.3	LLWU_ClearExternalWakeupsPinFlag	390
21.8.4	LLWU_EnableInternalModuleInterruptWakeup	391
21.8.5	LLWU_GetInternalWakeupsModuleFlag	391
21.8.6	LLWU_SetPinFilterMode	391
21.8.7	LLWU_GetPinFilterFlag	392
21.8.8	LLWU_ClearPinFilterFlag	392
Chapter	LMEM: Local Memory Controller Cache Control Driver	
22.1	Overview	393
22.2	Descriptions	393
22.3	Function groups	393
22.3.1	Local Memory Processor Code Bus Cache Control	393
22.3.2	Local Memory Processor System Bus Cache Control	394
22.4	Macro Definition Documentation	396
22.4.1	FSL_LMEM_DRIVER_VERSION	396
22.4.2	LMEM_CACHE_LINE_SIZE	396
22.4.3	LMEM_CACHE_SIZE_ONEWAY	396
22.5	Enumeration Type Documentation	397
22.5.1	lmem_cache_mode_t	397

Contents

Section Number	Title	Page Number
22.5.2	lmem_cache_region_t	397
22.5.3	lmem_cache_line_command_t	397
22.6	Function Documentation	398
22.6.1	LMEM_EnableCodeCache	398
22.6.2	LMEM_EnableCodeWriteBuffer	399
22.6.3	LMEM_CodeCacheInvalidateAll	399
22.6.4	LMEM_CodeCachePushAll	399
22.6.5	LMEM_CodeCacheClearAll	399
22.6.6	LMEM_CodeCacheInvalidateLine	400
22.6.7	LMEM_CodeCacheInvalidateMultiLines	400
22.6.8	LMEM_CodeCachePushLine	400
22.6.9	LMEM_CodeCachePushMultiLines	401
22.6.10	LMEM_CodeCacheClearLine	401
22.6.11	LMEM_CodeCacheClearMultiLines	402
22.6.12	LMEM_CodeCacheDemoteRegion	402

Chapter LPTMR: Low-Power Timer

23.1	Overview	403
23.2	Function groups	403
23.2.1	Initialization and deinitialization	403
23.2.2	Timer period Operations	403
23.2.3	Start and Stop timer operations	403
23.2.4	Status	404
23.2.5	Interrupt	404
23.3	Typical use case	404
23.3.1	LPTMR tick example	404
23.4	Data Structure Documentation	406
23.4.1	struct lptmr_config_t	406
23.5	Enumeration Type Documentation	407
23.5.1	lptmr_pin_select_t	407
23.5.2	lptmr_pin_polarity_t	407
23.5.3	lptmr_timer_mode_t	407
23.5.4	lptmr_prescaler_glitch_value_t	408
23.5.5	lptmr_prescaler_clock_select_t	408
23.5.6	lptmr_interrupt_enable_t	408
23.5.7	lptmr_status_flags_t	409
23.6	Function Documentation	409
23.6.1	LPTMR_Init	409
23.6.2	LPTMR_Deinit	409

Contents

Section Number	Title	Page Number
23.6.3	LPTMR_GetDefaultConfig	409
23.6.4	LPTMR_EnableInterrupts	410
23.6.5	LPTMR_DisableInterrupts	410
23.6.6	LPTMR_GetEnabledInterrupts	410
23.6.7	LPTMR_GetStatusFlags	410
23.6.8	LPTMR_ClearStatusFlags	411
23.6.9	LPTMR_SetTimerPeriod	411
23.6.10	LPTMR_GetCurrentTimerCount	411
23.6.11	LPTMR_StartTimer	412
23.6.12	LPTMR_StopTimer	412

Chapter **LPUART: Low Power UART Driver**

24.1	Overview	413
24.2	LPUART Driver	414
24.2.1	Overview	414
24.2.2	Typical use case	414
24.2.3	Data Structure Documentation	418
24.2.4	Macro Definition Documentation	420
24.2.5	Typedef Documentation	420
24.2.6	Enumeration Type Documentation	420
24.2.7	Function Documentation	423
24.3	LPUART DMA Driver	438
24.3.1	Overview	438
24.3.2	Data Structure Documentation	438
24.3.3	Typedef Documentation	439
24.3.4	Function Documentation	439
24.4	LPUART eDMA Driver	443
24.4.1	Overview	443
24.4.2	Data Structure Documentation	444
24.4.3	Typedef Documentation	445
24.4.4	Function Documentation	445
24.5	LPUART μCOS/II Driver	449
24.5.1	Overview	449
24.5.2	Data Structure Documentation	449
24.5.3	Function Documentation	450
24.6	LPUART μCOS/III Driver	453
24.6.1	Overview	453
24.6.2	Data Structure Documentation	453
24.6.3	Function Documentation	454

Contents

Section Number	Title	Page Number
24.7	LPUART FreeRTOS Driver	457
24.7.1	Overview	457
24.7.2	Data Structure Documentation	457
24.7.3	Function Documentation	458
 Chapter 25 MPU: Memory Protection Unit		
25.1	Overview	461
25.2	Initialization and Deinitialization	461
25.3	Basic Control Operations	462
25.4	Data Structure Documentation	465
25.4.1	struct mpu.hardware_info_t	465
25.4.2	struct mpu.access_err_info_t	465
25.4.3	struct mpu.rwxrights_master_access_control_t	466
25.4.4	struct mpu.rwrights_master_access_control_t	466
25.4.5	struct mpu.region_config_t	467
25.4.6	struct mpu.config_t	468
25.5	Macro Definition Documentation	469
25.5.1	FSL_MPU_DRIVER_VERSION	469
25.5.2	MPU_REGION_RWXRIGHTS_MASTER_SHIFT	469
25.5.3	MPU_REGION_RWXRIGHTS_MASTER_MASK	469
25.5.4	MPU_REGION_RWXRIGHTS_MASTER_WIDTH	469
25.5.5	MPU_REGION_RWXRIGHTS_MASTER	469
25.5.6	MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT	469
25.5.7	MPU_REGION_RWXRIGHTS_MASTER_PE_MASK	469
25.5.8	MPU_REGION_RWXRIGHTS_MASTER_PE	469
25.5.9	MPU_REGION_RWRIGHTS_MASTER_SHIFT	469
25.5.10	MPU_REGION_RWRIGHTS_MASTER_MASK	469
25.5.11	MPU_REGION_RWRIGHTS_MASTER	469
25.5.12	MPU_SLAVE_PORT_NUM	469
25.5.13	MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX	469
25.6	Enumeration Type Documentation	470
25.6.1	mpu.region_total_num_t	470
25.6.2	mpu.slave_t	470
25.6.3	mpu.err_access_control_t	470
25.6.4	mpu.err_access_type_t	470
25.6.5	mpu.err_attributes_t	470
25.6.6	mpu.supervisor_access_rights_t	471
25.6.7	mpu.user_access_rights_t	471
25.7	Function Documentation	471

Contents

Section Number	Title	Page Number
25.7.1	MPU_Init	471
25.7.2	MPU_Deinit	471
25.7.3	MPU_Enable	472
25.7.4	MPU_RegionEnable	472
25.7.5	MPU_GetHardwareInfo	472
25.7.6	MPU_SetRegionConfig	473
25.7.7	MPU_SetRegionAddr	473
25.7.8	MPU_SetRegionRwxMasterAccessRights	473
25.7.9	MPU_SetRegionRwMasterAccessRights	474
25.7.10	MPU_GetSlavePortErrorStatus	474
25.7.11	MPU_GetDetailErrorAccessInfo	475

Chapter [PDB: Programmable Delay Block](#)

26.1	Overview	477
26.2	Typical use case	477
26.2.1	Working as basic PDB counter with a PDB interrupt.	477
26.2.2	Working with an additional trigger. The ADC trigger is used as an example.	478
26.3	Data Structure Documentation	482
26.3.1	struct pdb_config_t	482
26.3.2	struct pdb_adc_pretrigger_config_t	483
26.3.3	struct pdb_dac_trigger_config_t	483
26.4	Macro Definition Documentation	484
26.4.1	FSL_PDB_DRIVER_VERSION	484
26.5	Enumeration Type Documentation	484
26.5.1	_pdb_status_flags	484
26.5.2	_pdb_adc_pretrigger_flags	484
26.5.3	_pdb_interrupt_enable	484
26.5.4	pdb_load_value_mode_t	484
26.5.5	pdb_prescaler_divider_t	485
26.5.6	pdb_divider_multiplication_factor_t	485
26.5.7	pdb_trigger_input_source_t	485
26.6	Function Documentation	486
26.6.1	PDB_Init	486
26.6.2	PDB_Deinit	486
26.6.3	PDB_GetDefaultConfig	487
26.6.4	PDB_Enable	487
26.6.5	PDB_DoSoftwareTrigger	487
26.6.6	PDB_DoLoadValues	487
26.6.7	PDB_EnableDMA	488

Contents

Section Number	Title	Page Number
26.6.8	PDB_EnableInterrupts	488
26.6.9	PDB_DisableInterrupts	488
26.6.10	PDB_GetStatusFlags	488
26.6.11	PDB_ClearStatusFlags	489
26.6.12	PDB_SetModulusValue	489
26.6.13	PDB_GetCounterValue	489
26.6.14	PDB_SetCounterDelayValue	489
26.6.15	PDB_SetADCPreTriggerConfig	490
26.6.16	PDB_SetADCPreTriggerDelayValue	490
26.6.17	PDB_GetADCPreTriggerStatusFlags	490
26.6.18	PDB_ClearADCPreTriggerStatusFlags	491
26.6.19	PDB_SetDACTriggerConfig	491
26.6.20	PDB_SetDACTriggerIntervalValue	491
26.6.21	PDB_EnablePulseOutTrigger	492
26.6.22	PDB_SetPulseOutTriggerDelayValue	492

Chapter PIT: Periodic Interrupt Timer

27.1	Overview	493
27.2	Function groups	493
27.2.1	Initialization and deinitialization	493
27.2.2	Timer period Operations	493
27.2.3	Start and Stop timer operations	493
27.2.4	Status	494
27.2.5	Interrupt	494
27.3	Typical use case	494
27.3.1	PIT tick example	494
27.4	Data Structure Documentation	496
27.4.1	struct pit_config_t	496
27.5	Enumeration Type Documentation	496
27.5.1	pit_chnl_t	496
27.5.2	pit_interrupt_enable_t	497
27.5.3	pit_status_flags_t	497
27.6	Function Documentation	497
27.6.1	PIT_Init	497
27.6.2	PIT_Deinit	497
27.6.3	PIT_GetDefaultConfig	497
27.6.4	PIT_SetTimerChainMode	498
27.6.5	PIT_EnableInterrupts	498
27.6.6	PIT_DisableInterrupts	498

Contents

Section Number	Title	Page Number
27.6.7	PIT_GetEnabledInterrupts	499
27.6.8	PIT_GetStatusFlags	499
27.6.9	PIT_ClearStatusFlags	499
27.6.10	PIT_SetTimerPeriod	500
27.6.11	PIT_GetCurrentTimerCount	500
27.6.12	PIT_StartTimer	501
27.6.13	PIT_StopTimer	501
27.6.14	PIT_GetLifetimeTimerCount	501

Chapter **PMC: Power Management Controller**

28.1	Overview	503
28.2	Data Structure Documentation	504
28.2.1	struct pmc_low_volt_detect_config_t	504
28.2.2	struct pmc_low_volt_warning_config_t	504
28.2.3	struct pmc_bandgap_buffer_config_t	504
28.3	Macro Definition Documentation	505
28.3.1	FSL_PMC_DRIVER_VERSION	505
28.4	Enumeration Type Documentation	505
28.4.1	pmc_low_volt_detect_volt_select_t	505
28.4.2	pmc_low_volt_warning_volt_select_t	505
28.5	Function Documentation	505
28.5.1	PMC_ConfigureLowVoltDetect	505
28.5.2	PMC_GetLowVoltDetectFlag	506
28.5.3	PMC_ClearLowVoltDetectFlag	506
28.5.4	PMC_ConfigureLowVoltWarning	506
28.5.5	PMC_GetLowVoltWarningFlag	507
28.5.6	PMC_ClearLowVoltWarningFlag	507
28.5.7	PMC_ConfigureBandgapBuffer	507
28.5.8	PMC_GetPeriphIOIsolationFlag	508
28.5.9	PMC_ClearPeriphIOIsolationFlag	508
28.5.10	PMC_IsRegulatorInRunRegulation	508

Chapter **PORT: Port Control and Interrupts**

29.1	Overview	511
29.2	Typical configuration use case	511
29.2.1	Input PORT configuration	511
29.2.2	I2C PORT Configuration	511
29.3	Data Structure Documentation	514

Contents

Section Number	Title	Page Number
29.3.1	struct port_digital_filter_config_t	514
29.3.2	struct port_pin_config_t	514
29.4	Macro Definition Documentation	514
29.4.1	FSL_PORT_DRIVER_VERSION	514
29.5	Enumeration Type Documentation	514
29.5.1	_port_pull	514
29.5.2	_port_slew_rate	515
29.5.3	_port_open_drain_enable	515
29.5.4	_port_passive_filter_enable	515
29.5.5	_port_drive_strength	515
29.5.6	_port_lock_register	515
29.5.7	port_mux_t	515
29.5.8	port_interrupt_t	516
29.5.9	port_digital_filter_clock_source_t	516
29.6	Function Documentation	516
29.6.1	PORT_SetPinConfig	516
29.6.2	PORT_SetMultiplePinsConfig	517
29.6.3	PORT_SetPinMux	517
29.6.4	PORT_EnablePinsDigitalFilter	518
29.6.5	PORT_SetDigitalFilterConfig	518
29.6.6	PORT_SetPinInterruptConfig	518
29.6.7	PORT_GetPinsInterruptFlags	519
29.6.8	PORT_ClearPinsInterruptFlags	520
Chapter RCM: Reset Control Module Driver		
30.1	Overview	523
30.2	Data Structure Documentation	524
30.2.1	struct rcm_reset_pin_filter_config_t	524
30.3	Macro Definition Documentation	524
30.3.1	FSL_RCM_DRIVER_VERSION	524
30.4	Enumeration Type Documentation	524
30.4.1	rcm_reset_source_t	524
30.4.2	rcm_run_wait_filter_mode_t	525
30.5	Function Documentation	525
30.5.1	RCM_GetPreviousResetSources	525
30.5.2	RCM_GetStickyResetSources	526
30.5.3	RCM_ClearStickyResetSources	526
30.5.4	RCM_ConfigureResetPinFilter	527

Contents

Section Number	Title	Page Number
30.5.5	RCM_GetEasyPortModePinStatus	527
Chapter	RNGA: Random Number Generator Accelerator Driver	
31.1	Overview	529
31.2	RNGA Initialization	529
31.3	Get random data from RNGA	529
31.4	RNGA Set/Get Working Mode	529
31.5	Seed RNGA	529
31.6	Macro Definition Documentation	531
31.6.1	FSL_RNGA_DRIVER_VERSION	531
31.7	Enumeration Type Documentation	531
31.7.1	rnga_mode_t	531
31.8	Function Documentation	531
31.8.1	RNGA_Init	531
31.8.2	RNGA_Deinit	531
31.8.3	RNGA_GetRandomData	532
31.8.4	RNGA_Seed	532
31.8.5	RNGA_SetMode	532
31.8.6	RNGA_GetMode	533
Chapter	RTC: Real Time Clock	
32.1	Overview	535
32.2	Function groups	535
32.2.1	Initialization and deinitialization	535
32.2.2	Set & Get Datetime	535
32.2.3	Set & Get Alarm	535
32.2.4	Start & Stop timer	536
32.2.5	Status	536
32.2.6	Interrupt	536
32.2.7	RTC Oscillator	536
32.2.8	Monotonic Counter	536
32.3	Typical use case	536
32.3.1	RTC tick example	536
32.4	Data Structure Documentation	540

Contents

Section Number	Title	Page Number
32.4.1	struct rtc_datetime_t	540
32.4.2	struct rtc_config_t	540
32.5	Enumeration Type Documentation	541
32.5.1	rtc_interrupt_enable_t	541
32.5.2	rtc_status_flags_t	541
32.5.3	rtc_osc_cap_load_t	541
32.6	Function Documentation	541
32.6.1	RTC_Init	541
32.6.2	RTC_Deinit	542
32.6.3	RTC_GetDefaultConfig	542
32.6.4	RTC_SetDatetime	542
32.6.5	RTC_GetDatetime	543
32.6.6	RTC_SetAlarm	543
32.6.7	RTC_GetAlarm	543
32.6.8	RTC_EnableInterrupts	544
32.6.9	RTC_DisableInterrupts	544
32.6.10	RTC_GetEnabledInterrupts	544
32.6.11	RTC_GetStatusFlags	544
32.6.12	RTC_ClearStatusFlags	545
32.6.13	RTC_StartTimer	545
32.6.14	RTC_StopTimer	545
32.6.15	RTC_SetOscCapLoad	545
32.6.16	RTC_Reset	546
32.6.17	RTC_GetMonotonicCounter	546
32.6.18	RTC_SetMonotonicCounter	546
32.6.19	RTC_IncrementMonotonicCounter	546
Chapter	SAI: Serial Audio Interface	
33.1	Overview	549
33.2	Typical use case	549
33.2.1	SAI Send/receive using an interrupt method	549
33.2.2	SAI Send/receive using a DMA method	550
33.3	Data Structure Documentation	556
33.3.1	struct sai_config_t	556
33.3.2	struct sai_transfer_format_t	556
33.3.3	struct sai_transfer_t	556
33.3.4	struct _sai_handle	557
33.4	Macro Definition Documentation	557
33.4.1	SAI_XFER_QUEUE_SIZE	557

Contents

Section Number	Title	Page Number
33.5	Enumeration Type Documentation	557
33.5.1	_sai_status_t	557
33.5.2	sai_protocol_t	558
33.5.3	sai_master_slave_t	558
33.5.4	sai_mono_stereo_t	558
33.5.5	sai_sync_mode_t	558
33.5.6	sai_mclk_source_t	559
33.5.7	sai_bclk_source_t	559
33.5.8	_sai_interrupt_enable_t	559
33.5.9	_sai_dma_enable_t	559
33.5.10	_sai_flags	559
33.5.11	sai_reset_type_t	560
33.5.12	sai_fifo_packing_t	560
33.5.13	sai_sample_rate_t	560
33.5.14	sai_word_width_t	560
33.6	Function Documentation	561
33.6.1	SAI_TxInit	561
33.6.2	SAI_RxInit	561
33.6.3	SAI_TxGetDefaultConfig	561
33.6.4	SAI_RxGetDefaultConfig	562
33.6.5	SAI_Deinit	562
33.6.6	SAI_TxReset	562
33.6.7	SAI_RxReset	562
33.6.8	SAI_TxEnable	563
33.6.9	SAI_RxEnable	563
33.6.10	SAI_TxGetStatusFlag	563
33.6.11	SAI_TxClearStatusFlags	563
33.6.12	SAI_RxGetStatusFlag	564
33.6.13	SAI_RxClearStatusFlags	564
33.6.14	SAI_TxEnableInterrupts	564
33.6.15	SAI_RxEnableInterrupts	565
33.6.16	SAI_TxDisableInterrupts	566
33.6.17	SAI_RxDisableInterrupts	566
33.6.18	SAI_TxEnableDMA	567
33.6.19	SAI_RxEnableDMA	567
33.6.20	SAI_TxGetDataRegisterAddress	567
33.6.21	SAI_RxGetDataRegisterAddress	568
33.6.22	SAI_TxSetFormat	568
33.6.23	SAI_RxSetFormat	569
33.6.24	SAI_WriteBlocking	569
33.6.25	SAI_WriteData	569
33.6.26	SAI_ReadBlocking	570
33.6.27	SAI_ReadData	570
33.6.28	SAI_TransferTxCreateHandle	570

Contents

Section Number	Title	Page Number
33.6.29	SAI_TransferRxCreateHandle	571
33.6.30	SAI_TransferTxSetFormat	571
33.6.31	SAI_TransferRxSetFormat	572
33.6.32	SAI_TransferSendNonBlocking	572
33.6.33	SAI_TransferReceiveNonBlocking	573
33.6.34	SAI_TransferGetSendCount	573
33.6.35	SAI_TransferGetReceiveCount	574
33.6.36	SAI_TransferAbortSend	574
33.6.37	SAI_TransferAbortReceive	575
33.6.38	SAI_TransferTxHandleIRQ	575
33.6.39	SAI_TransferRxHandleIRQ	575
33.7	SAI DMA Driver	576
33.7.1	Overview	576
33.7.2	Data Structure Documentation	577
33.7.3	Function Documentation	577
33.8	SAI eDMA Driver	583
33.8.1	Overview	583
33.8.2	Data Structure Documentation	584
33.8.3	Function Documentation	585

Chapter **SDHC: Secured Digital Host Controller Driver**

34.1	Overview	591
34.2	Typical use case	591
34.2.1	SDHC Operation	591
34.3	Data Structure Documentation	599
34.3.1	struct sdhc_adma2_descriptor_t	599
34.3.2	struct sdhc_capability_t	600
34.3.3	struct sdhc_transfer_config_t	600
34.3.4	struct sdhc_boot_config_t	600
34.3.5	struct sdhc_config_t	601
34.3.6	struct sdhc_data_t	602
34.3.7	struct sdhc_command_t	602
34.3.8	struct sdhc_transfer_t	602
34.3.9	struct sdhc_transfer_callback_t	603
34.3.10	struct _sdhc_handle	603
34.3.11	struct sdhc_host_t	604
34.4	Macro Definition Documentation	604
34.4.1	FSL_SDHC_DRIVER_VERSION	604
34.5	Typedef Documentation	604

Contents

Section Number	Title	Page Number
34.5.1	sdhc_adma1_descriptor_t	604
34.5.2	sdhc_transfer_function_t	604
34.6	Enumeration Type Documentation	604
34.6.1	_sdhc_status	604
34.6.2	_sdhc_capability_flag	604
34.6.3	_sdhc_wakeup_event	605
34.6.4	_sdhc_reset	605
34.6.5	_sdhc_transfer_flag	605
34.6.6	_sdhc_present_status_flag	606
34.6.7	_sdhc_interrupt_status_flag	606
34.6.8	_sdhc_auto_command12_error_status_flag	607
34.6.9	_sdhc_adma_error_status_flag	607
34.6.10	sdhc_adma_error_state_t	607
34.6.11	_sdhc_force_event	608
34.6.12	sdhc_data_bus_width_t	608
34.6.13	sdhc_endian_mode_t	608
34.6.14	sdhc_dma_mode_t	608
34.6.15	_sdhc_sdio_control_flag	609
34.6.16	sdhc_boot_mode_t	609
34.6.17	sdhc_command_type_t	609
34.6.18	sdhc_response_type_t	609
34.6.19	_sdhc_adma1_descriptor_flag	610
34.6.20	_sdhc_adma2_descriptor_flag	610
34.7	Function Documentation	610
34.7.1	SDHC_Init	610
34.7.2	SDHC_Deinit	611
34.7.3	SDHC_Reset	611
34.7.4	SDHC_SetAdmaTableConfig	611
34.7.5	SDHC_EnableInterruptStatus	612
34.7.6	SDHC_DisableInterruptStatus	612
34.7.7	SDHC_EnableInterruptSignal	612
34.7.8	SDHC_DisableInterruptSignal	613
34.7.9	SDHC_GetInterruptStatusFlags	613
34.7.10	SDHC_ClearInterruptStatusFlags	613
34.7.11	SDHC_GetAutoCommand12ErrorStatusFlags	613
34.7.12	SDHC_GetAdmaErrorStatusFlags	614
34.7.13	SDHC_GetPresentStatusFlags	614
34.7.14	SDHC_GetCapability	614
34.7.15	SDHC_EnableSdClock	615
34.7.16	SDHC_SetSdClock	615
34.7.17	SDHC_SetCardActive	615
34.7.18	SDHC_SetDataBusWidth	616
34.7.19	SDHC_SetTransferConfig	616

Contents

Section Number	Title	Page Number
34.7.20	SDHC_GetCommandResponse	617
34.7.21	SDHC_WriteData	617
34.7.22	SDHC_ReadData	617
34.7.23	SDHC_EnableWakeUpEvent	618
34.7.24	SDHC_EnableCardDetectTest	619
34.7.25	SDHC_SetCardDetectTestLevel	619
34.7.26	SDHC_EnableSdioControl	619
34.7.27	SDHC_SetContinueRequest	620
34.7.28	SDHC_SetMmcBootConfig	620
34.7.29	SDHC_SetForceEvent	620
34.7.30	SDHC_TransferBlocking	621
34.7.31	SDHC_TransferCreateHandle	621
34.7.32	SDHC_TransferNonBlocking	622
34.7.33	SDHC_TransferHandleIRQ	623

Chapter [SDRAMC: Synchronous DRAM Controller Driver](#)

35.1	Overview	625
35.2	Typical use case	625
35.3	Data Structure Documentation	628
35.3.1	struct sdramc_blockctl_config_t	628
35.3.2	struct sdramc_refresh_config_t	628
35.3.3	struct sdramc_config_t	629
35.4	Macro Definition Documentation	629
35.4.1	FSL_SDRAMC_DRIVER_VERSION	629
35.5	Enumeration Type Documentation	629
35.5.1	sdramc_refresh_time_t	629
35.5.2	sdramc_latency_t	630
35.5.3	sdramc_command_bit_location_t	630
35.5.4	sdramc_command_t	630
35.5.5	sdramc_port_size_t	631
35.5.6	sdramc_block_selection_t	631
35.6	Function Documentation	631
35.6.1	SDRAMC_Init	631
35.6.2	SDRAMC_Deinit	632
35.6.3	SDRAMC_SendCommand	632
35.6.4	SDRAMC_EnableWriteProtect	632
35.6.5	SDRAMC_EnableOperateValid	633

Contents

Section Number	Title	Page Number
Chapter	SIM: System Integration Module Driver	
36.1	Overview	635
36.2	Data Structure Documentation	635
36.2.1	struct sim_uid_t	635
36.3	Enumeration Type Documentation	636
36.3.1	_sim_usb_volt_reg_enable_mode	636
36.3.2	_sim_flash_mode	636
36.4	Function Documentation	636
36.4.1	SIM_SetUsbVoltRegulatorEnableMode	636
36.4.2	SIM_GetUniqueId	637
36.4.3	SIM_SetFlashMode	637
Chapter	SMC: System Mode Controller Driver	
37.1	Overview	639
37.2	Typical use case	639
37.2.1	Enter wait or stop modes	639
37.3	Data Structure Documentation	641
37.3.1	struct smc_power_mode_lls_config_t	641
37.3.2	struct smc_power_mode_vlls_config_t	641
37.4	Macro Definition Documentation	642
37.4.1	FSL_SMC_DRIVER_VERSION	642
37.5	Enumeration Type Documentation	642
37.5.1	smc_power_mode_protection_t	642
37.5.2	smc_power_state_t	642
37.5.3	smc_run_mode_t	642
37.5.4	smc_stop_mode_t	643
37.5.5	smc_stop_submode_t	643
37.5.6	smc_partial_stop_option_t	643
37.5.7	_smc_status	643
37.6	Function Documentation	643
37.6.1	SMC_SetPowerModeProtection	643
37.6.2	SMC_GetPowerModeState	645
37.6.3	SMC_PreEnterStopModes	645
37.6.4	SMC_PostExitStopModes	645
37.6.5	SMC_PreEnterWaitModes	645
37.6.6	SMC_PostExitWaitModes	645

Contents

Section Number	Title	Page Number
37.6.7	SMC_SetPowerModeRun	646
37.6.8	SMC_SetPowerModeHsrun	647
37.6.9	SMC_SetPowerModeWait	647
37.6.10	SMC_SetPowerModeStop	647
37.6.11	SMC_SetPowerModeVlpr	648
37.6.12	SMC_SetPowerModeVlpw	648
37.6.13	SMC_SetPowerModeVlps	648
37.6.14	SMC_SetPowerModeLls	648
37.6.15	SMC_SetPowerModeVlls	649

Chapter [TPM: Timer PWM Module](#)

38.1	Overview	651
38.2	Typical use case	652
38.2.1	PWM output	652
38.3	Data Structure Documentation	656
38.3.1	struct tpm_chnl_pwm_signal_param_t	656
38.3.2	struct tpm_dual_edge_capture_param_t	657
38.3.3	struct tpm_phase_params_t	657
38.3.4	struct tpm_config_t	657
38.4	Enumeration Type Documentation	658
38.4.1	tpm_chnl_t	658
38.4.2	tpm_pwm_mode_t	659
38.4.3	tpm_pwm_level_select_t	659
38.4.4	tpm_trigger_select_t	659
38.4.5	tpm_trigger_source_t	659
38.4.6	tpm_output_compare_mode_t	659
38.4.7	tpm_input_capture_edge_t	660
38.4.8	tpm_quad_decode_mode_t	660
38.4.9	tpm_phase_polarity_t	660
38.4.10	tpm_clock_source_t	660
38.4.11	tpm_clock_prescale_t	661
38.4.12	tpm_interrupt_enable_t	661
38.4.13	tpm_status_flags_t	661
38.5	Function Documentation	661
38.5.1	TPM_Init	661
38.5.2	TPM_Deinit	662
38.5.3	TPM_GetDefaultConfig	662
38.5.4	TPM_SetupPwm	662
38.5.5	TPM_UpdatePwmDutycycle	663
38.5.6	TPM_UpdateChnlEdgeLevelSelect	663

Contents

Section Number	Title	Page Number
38.5.7	TPM_SetupInputCapture	664
38.5.8	TPM_SetupOutputCompare	664
38.5.9	TPM_SetupDualEdgeCapture	664
38.5.10	TPM_SetupQuadDecode	665
38.5.11	TPM_EnableInterrupts	665
38.5.12	TPM_DisableInterrupts	665
38.5.13	TPM_GetEnabledInterrupts	666
38.5.14	TPM_GetStatusFlags	666
38.5.15	TPM_ClearStatusFlags	666
38.5.16	TPM_StartTimer	667
38.5.17	TPM_StopTimer	667

Chapter **UART: Universal Asynchronous Receiver/Transmitter Driver**

39.1	Overview	669
39.2	UART Driver	670
39.2.1	Overview	670
39.2.2	Typical use case	670
39.2.3	Data Structure Documentation	678
39.2.4	Macro Definition Documentation	680
39.2.5	Typedef Documentation	680
39.2.6	Enumeration Type Documentation	680
39.2.7	Function Documentation	682
39.3	UART DMA Driver	695
39.3.1	Overview	695
39.3.2	Data Structure Documentation	695
39.3.3	Typedef Documentation	696
39.3.4	Function Documentation	696
39.4	UART eDMA Driver	700
39.4.1	Overview	700
39.4.2	Data Structure Documentation	700
39.4.3	Typedef Documentation	701
39.4.4	Function Documentation	701
39.5	UART FreeRTOS Driver	705
39.5.1	Overview	705
39.5.2	Data Structure Documentation	705
39.5.3	Function Documentation	706
39.6	UART μCOS/II Driver	708
39.6.1	Overview	708
39.6.2	Data Structure Documentation	708

Contents

Section Number	Title	Page Number
39.6.3	Function Documentation	709
39.7	UART µCOS/III Driver	711
39.7.1	Overview	711
39.7.2	Data Structure Documentation	711
39.7.3	Function Documentation	712
Chapter	VREF: Voltage Reference Driver	
40.1	Overview	715
40.2	Typical use case and example	715
40.3	Data Structure Documentation	716
40.3.1	struct vref_config_t	716
40.4	Macro Definition Documentation	716
40.4.1	FSL_VREF_DRIVER_VERSION	716
40.5	Enumeration Type Documentation	716
40.5.1	vref_buffer_mode_t	716
40.6	Function Documentation	716
40.6.1	VREF_Init	716
40.6.2	VREF_Deinit	717
40.6.3	VREF_GetDefaultConfig	717
40.6.4	VREF_SetTrimVal	717
40.6.5	VREF_GetTrimVal	718
Chapter	WDOG: Watchdog Timer Driver	
41.1	Overview	719
41.2	Typical use case	719
41.3	Data Structure Documentation	721
41.3.1	struct wdog_work_mode_t	721
41.3.2	struct wdog_config_t	721
41.3.3	struct wdog_test_config_t	722
41.4	Macro Definition Documentation	722
41.4.1	FSL_WDOG_DRIVER_VERSION	722
41.5	Enumeration Type Documentation	722
41.5.1	wdog_clock_source_t	722
41.5.2	wdog_clock_prescaler_t	722

Contents

Section Number	Title	Page Number
41.5.3	wdog_test_mode_t	723
41.5.4	wdog_tested_byte_t	723
41.5.5	_wdog_interrupt_enable_t	723
41.5.6	_wdog_status_flags_t	723
41.6	Function Documentation	723
41.6.1	WDOG_GetDefaultConfig	723
41.6.2	WDOG_Init	724
41.6.3	WDOG_Deinit	724
41.6.4	WDOG_SetTestModeConfig	725
41.6.5	WDOG_Enable	725
41.6.6	WDOG_Disable	725
41.6.7	WDOG_EnableInterrupts	726
41.6.8	WDOG_DisableInterrupts	726
41.6.9	WDOG_GetStatusFlags	726
41.6.10	WDOG_ClearStatusFlags	727
41.6.11	WDOG_SetTimeoutValue	727
41.6.12	WDOG_SetWindowValue	728
41.6.13	WDOG_Unlock	728
41.6.14	WDOG_Refresh	728
41.6.15	WDOG_GetResetCount	729
41.6.16	WDOG_ClearResetCount	730
Chapter	Clock Driver	
42.1	Overview	731
42.2	Get frequency	731
42.3	External clock frequency	731
42.4	Data Structure Documentation	740
42.4.1	struct sim_clock_config_t	740
42.4.2	struct oscer_config_t	741
42.4.3	struct osc_config_t	741
42.4.4	struct mcg_pll_config_t	742
42.4.5	struct mcg_config_t	742
42.5	Macro Definition Documentation	743
42.5.1	FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL	743
42.5.2	FSL_CLOCK_DRIVER_VERSION	744
42.5.3	MCG_INTERNAL_IRC_48M	744
42.5.4	DMAMUX_CLOCKS	744
42.5.5	RTC_CLOCKS	744
42.5.6	ENET_CLOCKS	744

Contents

Section Number	Title	Page Number
42.5.7	PORT_CLOCKS	744
42.5.8	SAI_CLOCKS	744
42.5.9	FLEXBUS_CLOCKS	745
42.5.10	TSI_CLOCKS	745
42.5.11	LPUART_CLOCKS	745
42.5.12	EWM_CLOCKS	745
42.5.13	PIT_CLOCKS	745
42.5.14	DSPI_CLOCKS	746
42.5.15	LPTMR_CLOCKS	746
42.5.16	SDHC_CLOCKS	746
42.5.17	FTM_CLOCKS	746
42.5.18	EDMA_CLOCKS	746
42.5.19	FLEXCAN_CLOCKS	747
42.5.20	DAC_CLOCKS	747
42.5.21	ADC16_CLOCKS	747
42.5.22	SDRAM_CLOCKS	747
42.5.23	MPU_CLOCKS	747
42.5.24	VREF_CLOCKS	748
42.5.25	CMT_CLOCKS	748
42.5.26	UART_CLOCKS	748
42.5.27	TPM_CLOCKS	748
42.5.28	RNGA_CLOCKS	748
42.5.29	CRC_CLOCKS	749
42.5.30	I2C_CLOCKS	749
42.5.31	PDB_CLOCKS	749
42.5.32	FTF_CLOCKS	749
42.5.33	CMP_CLOCKS	749
42.5.34	SYS_CLK	750
42.6	Enumeration Type Documentation	750
42.6.1	clock_name_t	750
42.6.2	clock_usb_src_t	750
42.6.3	clock_usb_phy_src_t	750
42.6.4	clock_usb_pfd_src_t	751
42.6.5	clock_ip_name_t	751
42.6.6	osc_mode_t	751
42.6.7	_osc_cap_load	751
42.6.8	_oscer_enable_mode	751
42.6.9	mcg_fll_src_t	751
42.6.10	mcg_irc_mode_t	752
42.6.11	mcg_dmx32_t	752
42.6.12	mcg_drs_t	752
42.6.13	mcg_pll_ref_src_t	752
42.6.14	mcg_elkout_src_t	752
42.6.15	mcg_atm_select_t	753

Contents

Section Number	Title	Page Number
42.6.16	mcg_oscsel_t	753
42.6.17	mcg_pll_clk_select_t	753
42.6.18	mcg_monitor_mode_t	753
42.6.19	_mcg_status	753
42.6.20	_mcg_status_flags_t	754
42.6.21	_mcg_irclk_enable_mode	754
42.6.22	_mcg_pll_enable_mode	754
42.6.23	mcg_mode_t	754
42.7	Function Documentation	755
42.7.1	CLOCK_EnableClock	755
42.7.2	CLOCK_DisableClock	756
42.7.3	CLOCK_SetEr32kClock	756
42.7.4	CLOCK_SetSdhc0Clock	756
42.7.5	CLOCK_SetEnetTime0Clock	756
42.7.6	CLOCK_SetRmii0Clock	756
42.7.7	CLOCK_SetLpuartClock	757
42.7.8	CLOCK_SetTpmClock	757
42.7.9	CLOCK_SetTraceClock	757
42.7.10	CLOCK_SetPllFllSelClock	757
42.7.11	CLOCK_SetClkOutClock	757
42.7.12	CLOCK_SetRtcClkOutClock	758
42.7.13	CLOCK_EnableUsbhs0Clock	758
42.7.14	CLOCK_DisableUsbhs0Clock	758
42.7.15	CLOCK_EnableUsbhs0PhyPllClock	758
42.7.16	CLOCK_DisableUsbhs0PhyPllClock	759
42.7.17	CLOCK_EnableUsbhs0PfdClock	759
42.7.18	CLOCK_DisableUsbhs0PfdClock	759
42.7.19	CLOCK_EnableUsbfs0Clock	759
42.7.20	CLOCK_DisableUsbfs0Clock	760
42.7.21	CLOCK_SetOutDiv	760
42.7.22	CLOCK_GetFreq	760
42.7.23	CLOCK_GetCoreSysClkFreq	761
42.7.24	CLOCK_GetPlatClkFreq	761
42.7.25	CLOCK_GetBusClkFreq	761
42.7.26	CLOCK_GetFlexBusClkFreq	761
42.7.27	CLOCK_GetFlashClkFreq	761
42.7.28	CLOCK_GetPllFllSelClkFreq	762
42.7.29	CLOCK_GetEr32kClkFreq	762
42.7.30	CLOCK_GetOsc0ErClkFreq	762
42.7.31	CLOCK_GetOsc0ErClkUndivFreq	762
42.7.32	CLOCK_SetSimConfig	762
42.7.33	CLOCK_SetSimSafeDivs	762
42.7.34	CLOCK_GetOutClkFreq	763
42.7.35	CLOCK_GetFllFreq	763

Contents

Section Number	Title	Page Number
42.7.36	CLOCK_GetInternalRefClkFreq	763
42.7.37	CLOCK_GetFixedFreqClkFreq	763
42.7.38	CLOCK_GetPll0Freq	764
42.7.39	CLOCK_GetExtPllFreq	764
42.7.40	CLOCK_SetExtPllFreq	764
42.7.41	CLOCK_SetLowPowerEnable	764
42.7.42	CLOCK_SetInternalRefClkConfig	765
42.7.43	CLOCK_SetExternalRefClkConfig	765
42.7.44	CLOCK_SetFllExtRefDiv	766
42.7.45	CLOCK_EnablePll0	766
42.7.46	CLOCK_DisablePll0	766
42.7.47	CLOCK_CalcPllDiv	766
42.7.48	CLOCK_SetPllClkSel	767
42.7.49	CLOCK_SetOsc0MonitorMode	767
42.7.50	CLOCK_SetRtcOscMonitorMode	767
42.7.51	CLOCK_SetPll0MonitorMode	767
42.7.52	CLOCK_SetExtPllMonitorMode	768
42.7.53	CLOCK_GetStatusFlags	768
42.7.54	CLOCK_ClearStatusFlags	768
42.7.55	OSC_SetExtRefClkConfig	769
42.7.56	OSC_SetCapLoad	769
42.7.57	CLOCK_InitOsc0	769
42.7.58	CLOCK_DeinitOsc0	770
42.7.59	CLOCK_SetXtal0Freq	770
42.7.60	CLOCK_SetXtal32Freq	770
42.7.61	CLOCK_TrimInternalRefClk	770
42.7.62	CLOCK_GetMode	771
42.7.63	CLOCK_SetFeiMode	771
42.7.64	CLOCK_SetFeeMode	772
42.7.65	CLOCK_SetFbiMode	772
42.7.66	CLOCK_SetFbeMode	773
42.7.67	CLOCK_SetBlpiMode	774
42.7.68	CLOCK_SetBlpeMode	774
42.7.69	CLOCK_SetPbeMode	774
42.7.70	CLOCK_SetPeeMode	775
42.7.71	CLOCK_ExternalModeToFbeModeQuick	775
42.7.72	CLOCK_InternalModeToFbiModeQuick	776
42.7.73	CLOCK_BootToFeiMode	776
42.7.74	CLOCK_BootToFeeMode	777
42.7.75	CLOCK_BootToBlpiMode	777
42.7.76	CLOCK_BootToBlpeMode	778
42.7.77	CLOCK_BootToPbeMode	778
42.7.78	CLOCK_SetMcgConfig	779
42.8	Variable Documentation	779

Contents

Section Number	Title	Page Number
42.8.1	g_xtal0Freq	779
42.8.2	g_xtal32Freq	780
42.9	Multipurpose Clock Generator (MCG)	781
42.9.1	Function description	781
42.9.2	Typical use case	783
Chapter Debug Console		
43.1	Overview	787
43.2	Function groups	787
43.2.1	Initialization	787
43.2.2	Advanced Feature	788
43.3	Typical use case	791
43.4	Semihosting	793
43.4.1	Guide Semihosting for IAR	793
43.4.2	Guide Semihosting for Keil µVision	793
43.4.3	Guide Semihosting for KDS	795
43.4.4	Guide Semihosting for ATL	795
43.4.5	Guide Semihosting for ARMGCC	796
Chapter Notification Framework		
44.1	Overview	799
44.2	Notifier Overview	799
44.3	Data Structure Documentation	801
44.3.1	struct notifier_notification_block_t	801
44.3.2	struct notifier_callback_config_t	802
44.3.3	struct notifier_handle_t	802
44.4	Typedef Documentation	803
44.4.1	notifier_user_config_t	803
44.4.2	notifier_user_function_t	803
44.4.3	notifier_callback_t	804
44.5	Enumeration Type Documentation	804
44.5.1	_notifier_status	804
44.5.2	notifier_policy_t	805
44.5.3	notifier_notification_type_t	805
44.5.4	notifier_callback_type_t	805

Contents

Section Number	Title	Page Number
44.6	Function Documentation	806
44.6.1	NOTIFIER_CreateHandle	806
44.6.2	NOTIFIER_SwitchConfig	807
44.6.3	NOTIFIER_GetErrorCallbackIndex	808
 Chapter Shell		
45.1	Overview	809
45.2	Function groups	809
45.2.1	Initialization	809
45.2.2	Advanced Feature	809
45.2.3	Shell Operation	810
45.3	Data Structure Documentation	811
45.3.1	struct shell_context_struct	811
45.3.2	struct shell_command_context_t	812
45.3.3	struct shell_command_context_list_t	812
45.4	Macro Definition Documentation	813
45.4.1	SHELL_USE_HISTORY	813
45.4.2	SHELL_SEARCH_IN_HIST	813
45.4.3	SHELL_USE_FILE_STREAM	813
45.4.4	SHELL_AUTO_COMPLETE	813
45.4.5	SHELL_BUFFER_SIZE	813
45.4.6	SHELL_MAX_ARGS	813
45.4.7	SHELL_HIST_MAX	813
45.4.8	SHELL_MAX_CMD	813
45.5	Typedef Documentation	813
45.5.1	send_data_cb_t	813
45.5.2	recv_data_cb_t	813
45.5.3	printf_data_t	813
45.5.4	cmd_function_t	813
45.6	Enumeration Type Documentation	813
45.6.1	fun_key_status_t	813
45.7	Function Documentation	814
45.7.1	SHELL_Init	814
45.7.2	SHELL_RegisterCommand	814
45.7.3	SHELL_Main	814

Chapter 1

Introduction

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

The SDK v2.1 is built with the following runtime software components:

- ARM® and DSP standard libraries, and CMSIS-compliant device header files which provide direct access to the peripheral registers.
- Peripheral drivers that provide stateless, high-performance, ease-of-use APIs. Communication drivers provide higher-level transactional APIs for a higher-performance option.
- RTOS wrapper driver built on top of KSDK peripheral drivers and leverage native RTOS services to better comply to the RTOS cases.
- Real time operation systems (RTOS) including FreeRTOS OS, μC/OS-II, and μC/OS-III.
- Stacks and middleware in source or object formats including:
 - A USB device, host, and OTG stack with comprehensive USB class support.
 - CMSIS-DSP, a suite of common signal processing functions.
 - The SDK v2.1 comes complete with software examples demonstrating the usage of the peripheral drivers, RTOS wrapper drivers, middleware and RTOSes.

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

- IAR Embedded Workbench
- Keil MDK
- LPCXpresso IDE

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

The rest of this document describes the API references in detail for the peripheral drivers and RTOS wrapper drivers. For the latest version of this and other Kinetis SDK documents, see the kex.nxp.com/apidoc.

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

Table 2: KSDK Folder Structure

Chapter 2

Driver errors status

- `kStatus_DSPI_Error` = 601
- `kStatus_EDMA_QueueFull` = 5100
- `kStatus_EDMA_Busy` = 5101
- `kStatus_ENET_RxFrameError` = 4000
- `kStatus_ENET_RxFrameFail` = 4001
- `kStatus_ENET_RxFrameEmpty` = 4002
- `kStatus_ENET_TxFrameBusy` = 4003
- `kStatus_ENET_TxFrameFail` = 4004
- `#kStatus_ENET_PtpTsRingFull` = 4005
- `#kStatus_ENET_PtpTsRingEmpty` = 4006
- `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

Chapter 3

Architectural Overview

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

Overview

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

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

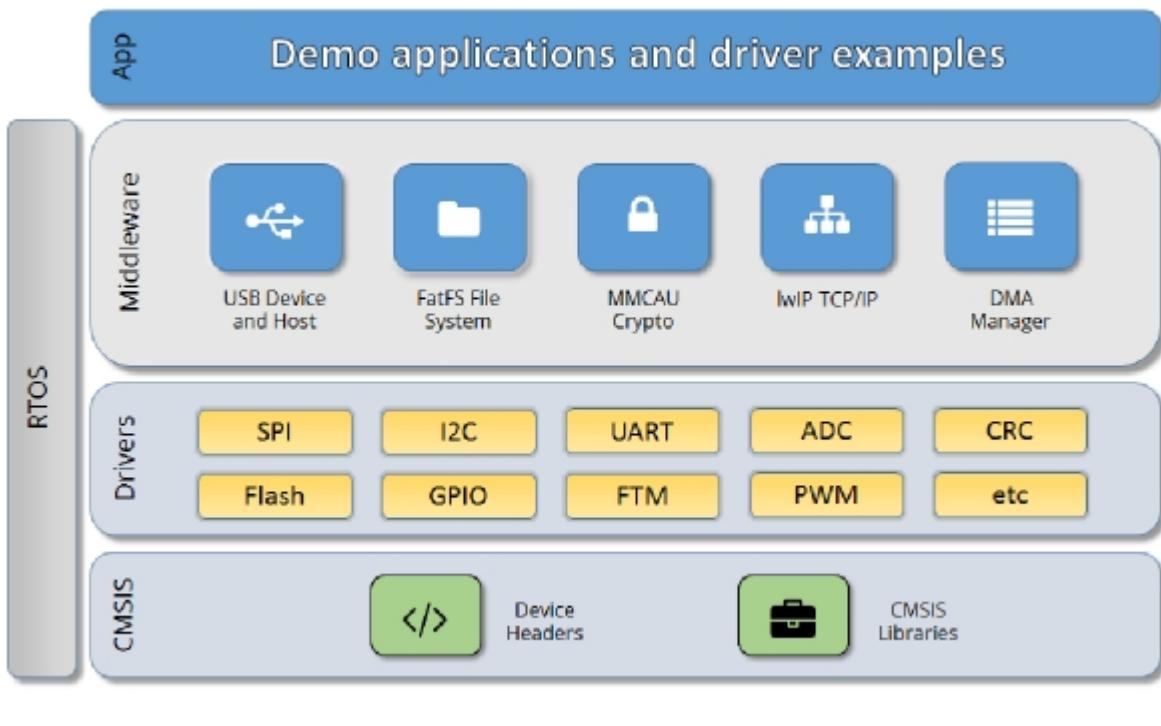


Figure 1: KSDK Block Diagram

Kinetis MCU header files

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

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

CMSIS Support

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

KSDK Peripheral Drivers

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

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

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

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

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

Interrupt handling for transactional APIs

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

```
PUBWEAK SPI0_IRQHandler  
PUBWEAK SPI0_DriverIRQHandler  
SPI0_IRQHandler
```

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

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

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

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

Feature Header Files

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

Application

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



Chapter 4

Trademarks

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

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

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

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

© 2016 Freescale Semiconductors, Inc.

Chapter 5

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

5.1 Overview

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

5.2 Typical use case

5.2.1 Polling Configuration

```
adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

5.2.2 Interrupt Configuration

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

Typical use case

```
// ...

adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

// ...

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

Data Structures

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

Enumerations

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

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

 `kADC16_ActiveFlag` = ADC_SC2_ADACT_MASK,

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

 `kADC16_ChannelMuxA` = 0U,

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

 `kADC16_ClockDivider1` = 0U,

 `kADC16_ClockDivider2` = 1U,

 `kADC16_ClockDivider4` = 2U,

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

 `kADC16_Resolution8or9Bit` = 0U,

 `kADC16_Resolution12or13Bit` = 1U,

 `kADC16_Resolution10or11Bit` = 2U,

 `kADC16_ResolutionSE8Bit` = `kADC16_Resolution8or9Bit`,

 `kADC16_ResolutionSE12Bit` = `kADC16_Resolution12or13Bit`,

 `kADC16_ResolutionSE10Bit` = `kADC16_Resolution10or11Bit`,

 `kADC16_ResolutionDF9Bit` = `kADC16_Resolution8or9Bit`,

 `kADC16_ResolutionDF13Bit` = `kADC16_Resolution12or13Bit`,

 `kADC16_ResolutionDF11Bit` = `kADC16_Resolution10or11Bit`,

 `kADC16_Resolution16Bit` = 3U,

 `kADC16_ResolutionSE16Bit` = `kADC16_Resolution16Bit`,

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

 `kADC16_ClockSourceAlt0` = 0U,

 `kADC16_ClockSourceAlt1` = 1U,

 `kADC16_ClockSourceAlt2` = 2U,

 `kADC16_ClockSourceAlt3` = 3U,

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

 `kADC16_LongSampleCycle24` = 0U,

 `kADC16_LongSampleCycle16` = 1U,

 `kADC16_LongSampleCycle10` = 2U,

 `kADC16_LongSampleCycle6` = 3U,

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

 `kADC16_ReferenceVoltageSourceVref` = 0U,

 `kADC16_ReferenceVoltageSourceValt` = 1U }

Typical use case

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

Driver version

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

Initialization

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

Advanced Features

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

Conversion Channel

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

5.3 Data Structure Documentation

5.3.1 struct adc16_config_t

Data Fields

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

Data Structure Documentation

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 `adc16_reference_voltage_source_t adc16_config_t::referenceVoltageSource`

5.3.1.0.0.1.2 `adc16_clock_source_t adc16_config_t::clockSource`

5.3.1.0.0.1.3 `bool adc16_config_t::enableAsynchronousClock`

5.3.1.0.0.1.4 `adc16_clock_divider_t adc16_config_t::clockDivider`

5.3.1.0.0.1.5 `adc16_resolution_t adc16_config_t::resolution`

5.3.1.0.0.1.6 `adc16_long_sample_mode_t adc16_config_t::longSampleMode`

5.3.1.0.0.1.7 `bool adc16_config_t::enableHighSpeed`

5.3.1.0.0.1.8 `bool adc16_config_t::enableLowPower`

5.3.1.0.0.1.9 `bool adc16_config_t::enableContinuousConversion`

5.3.2 struct `adc16_hardware_compare_config_t`

Data Fields

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

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 `adc16_hardware_compare_mode_t adc16_hardware_compare_config_t::hardwareCompareMode`

See "adc16_hardware_compare_mode_t".

5.3.2.0.0.2.2 `int16_t adc16_hardware_compare_config_t::value1`

5.3.2.0.0.2.3 `int16_t adc16_hardware_compare_config_t::value2`

5.3.3 struct `adc16_channel_config_t`

Data Fields

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

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

5.3.3.0.0.3 Field Documentation

5.3.3.0.0.3.1 `uint32_t adc16_channel_config_t::channelNumber`

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

5.3.3.0.0.3.2 `bool adc16_channel_config_t::enableInterruptOnConversionCompleted`

5.3.3.0.0.3.3 `bool adc16_channel_config_t::enableDifferentialConversion`

5.4 Macro Definition Documentation

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

5.5 Enumeration Type Documentation

5.5.1 `enum _adc16_channel_status_flags`

Enumerator

kADC16_ChannelConversionDoneFlag Conversion done.

5.5.2 `enum _adc16_status_flags`

Enumerator

kADC16_ActiveFlag Converter is active.

kADC16_CalibrationFailedFlag Calibration is failed.

5.5.3 `enum adc16_channel_mux_mode_t`

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

Enumerator

kADC16_ChannelMuxA For channel with channel mux a.

kADC16_ChannelMuxB For channel with channel mux b.

Enumeration Type Documentation

5.5.4 enum adc16_clock_divider_t

Enumerator

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

5.5.5 enum adc16_resolution_t

Enumerator

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

5.5.6 enum adc16_clock_source_t

Enumerator

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

5.5.7 enum adc16_long_sample_mode_t

Enumerator

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

kADC16_LongSampleCycle10 6 extra ADCK cycles, 10 ADCK cycles total.

kADC16_LongSampleCycle6 2 extra ADCK cycles, 6 ADCK cycles total.

kADC16_LongSampleDisabled Disable the long sample feature.

5.5.8 enum adc16_reference_voltage_source_t

Enumerator

kADC16_ReferenceVoltageSourceVref For external pins pair of VrefH and VrefL.

kADC16_ReferenceVoltageSourceValt For alternate reference pair of ValtH and ValtL.

5.5.9 enum adc16_hardware_average_mode_t

Enumerator

kADC16_HardwareAverageCount4 For hardware average with 4 samples.

kADC16_HardwareAverageCount8 For hardware average with 8 samples.

kADC16_HardwareAverageCount16 For hardware average with 16 samples.

kADC16_HardwareAverageCount32 For hardware average with 32 samples.

kADC16_HardwareAverageDisabled Disable the hardware average feature.

5.5.10 enum adc16_hardware_compare_mode_t

Enumerator

kADC16_HardwareCompareMode0 $x < \text{value1}$.

kADC16_HardwareCompareMode1 $x > \text{value1}$.

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

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

5.6 Function Documentation

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

Function Documentation

Parameters

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

5.6.2 void ADC16_Deinit (ADC_Type * *base*)

Parameters

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

5.6.3 void ADC16_GetDefaultConfig (adc16_config_t * *config*)

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

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

Parameters

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

5.6.4 status_t ADC16_DoAutoCalibration (ADC_Type * *base*)

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

Parameters

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

Returns

Execution status.

Return values

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

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

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

Parameters

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

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

Parameters

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

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

Function Documentation

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

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

Parameters

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

Returns

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

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

Parameters

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

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

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

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

Function Documentation

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

Parameters

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

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

Parameters

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

Returns

Conversion value.

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

Parameters

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

Returns

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

Function Documentation

Chapter 6

CMP: Analog Comparator Driver

6.1 Overview

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

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

6.2 Typical use case

6.2.1 Polling Configuration

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

    // ...

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

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

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

6.2.2 Interrupt Configuration

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

Typical use case

```
// ...

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

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

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

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

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

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

    while (1)
    {
    }
}
```

Data Structures

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

Enumerations

- enum [_cmp_interrupt_enable](#) {
 kCMP_OutputRisingInterruptEnable = CMP_SCR_IER_MASK,
 kCMP_OutputFallingInterruptEnable = CMP_SCR_IEF_MASK }

- *Interrupt enable/disable mask.*
- enum `_cmp_status_flags` {

 `kCMP_OutputRisingEventFlag` = CMP_SCR_CFR_MASK,
 `kCMP_OutputFallingEventFlag` = CMP_SCR_CFF_MASK,
 `kCMP_OutputAssertEventFlag` = CMP_SCR_COUT_MASK }

Status flags' mask.
- enum `cmp_hysteresis_mode_t` {

 `kCMP_HysteresisLevel0` = 0U,
 `kCMP_HysteresisLevel1` = 1U,
 `kCMP_HysteresisLevel2` = 2U,
 `kCMP_HysteresisLevel3` = 3U }

CMP Hysteresis mode.
- enum `cmp_reference_voltage_source_t` {

 `kCMP_VrefSourceVin1` = 0U,
 `kCMP_VrefSourceVin2` = 1U }

CMP Voltage Reference source.

Driver version

- #define `FSL_CMP_DRIVER_VERSION` (MAKE_VERSION(2, 0, 0))

CMP driver version 2.0.0.

Initialization

- void `CMP_Init` (CMP_Type *base, const `cmp_config_t` *config)

Initializes the CMP.
- void `CMP_Deinit` (CMP_Type *base)

De-initializes the CMP module.
- static void `CMP_Enable` (CMP_Type *base, bool enable)

Enables/disables the CMP module.
- void `CMP_GetDefaultConfig` (`cmp_config_t` *config)

Initializes the CMP user configuration structure.
- void `CMP_SetInputChannels` (CMP_Type *base, uint8_t positiveChannel, uint8_t negativeChannel)

Sets the input channels for the comparator.

Advanced Features

- void `CMP_EnableDMA` (CMP_Type *base, bool enable)

Enables/disables the DMA request for rising/falling events.
- static void `CMP_EnableWindowMode` (CMP_Type *base, bool enable)

Enables/disables the window mode.
- static void `CMP_EnablePassThroughMode` (CMP_Type *base, bool enable)

Enables/disables the pass through mode.
- void `CMP_SetFilterConfig` (CMP_Type *base, const `cmp_filter_config_t` *config)

Configures the filter.
- 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.

Data Structure Documentation

- void [CMP_DisableInterrupts](#) (CMP_Type *base, uint32_t mask)
Disables the interrupts.

Results

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

6.3 Data Structure Documentation

6.3.1 struct cmp_config_t

Data Fields

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

6.3.1.0.0.4 Field Documentation

6.3.1.0.0.4.1 bool cmp_config_t::enableCmp

6.3.1.0.0.4.2 cmp_hysteresis_mode_t cmp_config_t::hysteresisMode

6.3.1.0.0.4.3 bool cmp_config_t::enableHighSpeed

6.3.1.0.0.4.4 bool cmp_config_t::enableInvertOutput

6.3.1.0.0.4.5 bool cmp_config_t::useUnfilteredOutput

6.3.1.0.0.4.6 bool cmp_config_t::enablePinOut

6.3.1.0.0.4.7 bool cmp_config_t::enableTriggerMode

6.3.2 struct cmp_filter_config_t

Data Fields

- **bool enableSample**
Using the external SAMPLE as a sampling clock input or using a divided bus clock.
- **uint8_t filterCount**
Filter Sample Count.
- **uint8_t filterPeriod**
Filter Sample Period.

6.3.2.0.0.5 Field Documentation

6.3.2.0.0.5.1 bool cmp_filter_config_t::enableSample

6.3.2.0.0.5.2 uint8_t cmp_filter_config_t::filterCount

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

6.3.2.0.0.5.3 uint8_t cmp_filter_config_t::filterPeriod

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

6.3.3 struct cmp_dac_config_t

Data Fields

- **cmp_reference_voltage_source_t referenceVoltageSource**
Supply voltage reference source.
- **uint8_t DACValue**
Value for the DAC Output Voltage.

Enumeration Type Documentation

6.3.3.0.0.6 Field Documentation

6.3.3.0.0.6.1 `cmp_reference_voltage_source_t cmp_dac_config_t::referenceVoltageSource`

6.3.3.0.0.6.2 `uint8_t cmp_dac_config_t::DACValue`

Available range is 0-63.

6.4 Macro Definition Documentation

6.4.1 `#define FSL_CMP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

6.5 Enumeration Type Documentation

6.5.1 enum _cmp_interrupt_enable

Enumerator

kCMP_OutputRisingInterruptEnable Comparator interrupt enable rising.

kCMP_OutputFallingInterruptEnable Comparator interrupt enable falling.

6.5.2 enum _cmp_status_flags

Enumerator

kCMP_OutputRisingEventFlag Rising-edge on the comparison output has occurred.

kCMP_OutputFallingEventFlag Falling-edge on the comparison output has occurred.

kCMP_OutputAssertEventFlag Return the current value of the analog comparator output.

6.5.3 enum cmp_hysteresis_mode_t

Enumerator

kCMP_HysteresisLevel0 Hysteresis level 0.

kCMP_HysteresisLevel1 Hysteresis level 1.

kCMP_HysteresisLevel2 Hysteresis level 2.

kCMP_HysteresisLevel3 Hysteresis level 3.

6.5.4 enum cmp_reference_voltage_source_t

Enumerator

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

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

6.6 Function Documentation

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

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

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

Parameters

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

6.6.2 void CMP_Deinit (**CMP_Type** * *base*)

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

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

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

Parameters

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

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

Parameters

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

Function Documentation

<i>enable</i>	Enables or disables the module.
---------------	---------------------------------

6.6.4 void CMP_GetDefaultConfig (cmp_config_t * *config*)

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

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

Parameters

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

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

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

Parameters

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

6.6.6 void CMP_EnableDMA (CMP_Type * *base*, bool *enable*)

This function enables/disables the DMA request for rising/falling events. Either event triggers the generation of the DMA request from CMP if the DMA feature is enabled. Both events are ignored for generating the DMA request from the CMP if the DMA is disabled.

Parameters

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

**6.6.7 static void CMP_EnableWindowMode (*CMP_Type* * *base*, *bool enable*)
[inline], [static]**

Parameters

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

6.6.8 static void CMP_EnablePassThroughMode (*CMP_Type* * *base*, *bool enable*) [inline], [static]

Parameters

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

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

Parameters

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

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

Function Documentation

Parameters

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

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

Parameters

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

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

Parameters

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

6.6.13 uint32_t CMP_GetStatusFlags (CMP_Type * *base*)

Parameters

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

Returns

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

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

Parameters

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

Function Documentation

Chapter 7

CMT: Carrier Modulator Transmitter Driver

7.1 Overview

The carrier modulator transmitter (CMT) module provides the means to generate the protocol timing and carrier signals for a wide 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 KSDK provides a driver for the CMT module of the Kinetis devices.

7.2 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 (bus_clock_hz/8000000). The secondary prescaler is the "cmtDivider". The clocks for the CMT module are listed below.

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

7.3 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_IROCtlLow = 0U,
kCMT_IROCtlHigh = 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)

Data Structure Documentation

Disables the CMT module and gate control.

Basic Control Operations

- void [CMT_SetMode](#) (CMT_Type *base, [cmt_mode_t](#) mode, [cmt_modulate_config_t](#) *modulateConfig)
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_SetCarrierGenerateCountOne](#) (CMT_Type *base, uint32_t highCount, uint32_t lowCount)
Sets the primary data set for the CMT carrier generator counter.
- static void [CMT_SetCarrierGenerateCountTwo](#) (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.

7.4 Data Structure Documentation

7.4.1 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.

7.4.1.0.0.7 Field Documentation

- 7.4.1.0.0.7.1 `uint8_t cmt_modulate_config_t::highCount1`
- 7.4.1.0.0.7.2 `uint8_t cmt_modulate_config_t::lowCount1`
- 7.4.1.0.0.7.3 `uint8_t cmt_modulate_config_t::highCount2`
- 7.4.1.0.0.7.4 `uint8_t cmt_modulate_config_t::lowCount2`
- 7.4.1.0.0.7.5 `uint16_t cmt_modulate_config_t::markCount`
- 7.4.1.0.0.7.6 `uint16_t cmt_modulate_config_t::spaceCount`

7.4.2 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.

7.4.2.0.0.8 Field Documentation

- 7.4.2.0.0.8.1 `bool cmt_config_t::isInterruptEnabled`
- 7.4.2.0.0.8.2 `bool cmt_config_t::isIroEnabled`
- 7.4.2.0.0.8.3 `cmt_infrared_output_polarity_t cmt_config_t::iroPolarity`
- 7.4.2.0.0.8.4 `cmt_second_clkdiv_t cmt_config_t::divider`

7.5 Macro Definition Documentation

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

7.6 Enumeration Type Documentation

7.6.1 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.

7.6.2 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

kCMT_PrimaryClkDiv1 The intermediate frequency is the bus clock divided by 1.

kCMT_PrimaryClkDiv2 The intermediate frequency is the bus clock divided by 2.

kCMT_PrimaryClkDiv3 The intermediate frequency is the bus clock divided by 3.

kCMT_PrimaryClkDiv4 The intermediate frequency is the bus clock divided by 4.

kCMT_PrimaryClkDiv5 The intermediate frequency is the bus clock divided by 5.

kCMT_PrimaryClkDiv6 The intermediate frequency is the bus clock divided by 6.

kCMT_PrimaryClkDiv7 The intermediate frequency is the bus clock divided by 7.

kCMT_PrimaryClkDiv8 The intermediate frequency is the bus clock divided by 8.

kCMT_PrimaryClkDiv9 The intermediate frequency is the bus clock divided by 9.

kCMT_PrimaryClkDiv10 The intermediate frequency is the bus clock divided by 10.

kCMT_PrimaryClkDiv11 The intermediate frequency is the bus clock divided by 11.

kCMT_PrimaryClkDiv12 The intermediate frequency is the bus clock divided by 12.

kCMT_PrimaryClkDiv13 The intermediate frequency is the bus clock divided by 13.

kCMT_PrimaryClkDiv14 The intermediate frequency is the bus clock divided by 14.

kCMT_PrimaryClkDiv15 The intermediate frequency is the bus clock divided by 15.

kCMT_PrimaryClkDiv16 The intermediate frequency is the bus clock divided by 16.

7.6.3 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

kCMT_SecondClkDiv1 The CMT clock is the intermediate frequency frequency divided by 1.

kCMT_SecondClkDiv2 The CMT clock is the intermediate frequency frequency divided by 2.

kCMT_SecondClkDiv4 The CMT clock is the intermediate frequency frequency divided by 4.

kCMT_SecondClkDiv8 The CMT clock is the intermediate frequency frequency divided by 8.

7.6.4 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.

7.6.5 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.

7.6.6 enum _cmt_interrupt_enable

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

Enumerator

kCMT_EndOfCycleInterruptEnable CMT end of cycle interrupt.

7.7 Function Documentation

7.7.1 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.
---------------	--

7.7.2 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.

7.7.3 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.
-------------	------------------------------

7.7.4 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>modulateConfig</i>	The carrier generation and modulator configuration.

7.7.5 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.

7.7.6 `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.

7.7.7 static void CMT_SetCarrierGenerateCountOne (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 Tcmt, the period of the carrier generator signal equals (*highCount* + *lowCount*) * Tcmt. 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.

7.7.8 static void CMT_SetCarrierGenerateCountTwo (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 Tcmt, the period of the carrier generator signal equals (*highCount* + *lowCount*) * Tcmt. 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.

7.7.9 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 Fcmt and the carrier out signal frequency is fcg:

- In Time and Baseband mode: The mark period of the generated signal equals $(\text{markCount} + 1) / (\text{Fcmt}/8)$. The space period of the generated signal equals $\text{spaceCount} / (\text{Fcmt}/8)$.
- In FSK mode: The mark period of the generated signal equals $(\text{markCount} + 1)/\text{fcg}$. The space period of the generated signal equals $\text{spaceCount} / \text{fcg}$.

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.

7.7.10 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.

Function Documentation

7.7.11 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"

7.7.12 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 .

7.7.13 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 .

7.7.14 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 8

CRC: Cyclic Redundancy Check Driver

8.1 Overview

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

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

8.2 CRC Driver Initialization and Configuration

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

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

8.3 CRC Write Data

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

8.4 CRC Get Checksum

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

[CRC_Init\(\)](#) / [CRC_WriteData\(\)](#) / [CRC_Get16bitResult\(\)](#) to get the final checksum.

[CRC_Init\(\)](#) / [CRC_WriteData\(\)](#) / ... / [CRC_WriteData\(\)](#) / [CRC_Get16bitResult\(\)](#) to get the final checksum.

CRC 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.

8.5 Comments about API usage in RTOS

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

The triplets

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

The triplets are protected by the RTOS mutex to protect the CRC module against concurrent accesses from different tasks. This is an example.

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

8.6 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.

8.7 CRC Driver Examples

8.7.1 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);

```

8.7.2 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

8.8 Data Structure Documentation

8.8.1 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\(\)](#)

8.8.1.0.0.9 Field Documentation

8.8.1.0.0.9.1 uint32_t crc_config_t::polynomial

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

8.8.1.0.0.9.2 bool crc_config_t::reflectIn

8.8.1.0.0.9.3 bool crc_config_t::reflectOut

8.8.1.0.0.9.4 bool crc_config_t::complementChecksum

8.8.1.0.0.9.5 crc_bits_t crc_config_t::crcBits

8.9 Macro Definition Documentation

8.9.1 #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

8.9.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as default.

8.10 Enumeration Type Documentation

8.10.1 enum crc_bits_t

Enumerator

kCrcBits16 Generate 16-bit CRC code.

kCrcBits32 Generate 32-bit CRC code.

8.10.2 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.

8.11 Function Documentation

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

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

Parameters

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

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

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

Function Documentation

Parameters

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

8.11.3 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.
---------------	---------------------------------------

8.11.4 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.

8.11.5 *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.

8.11.6 `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.

Function Documentation

Chapter 9

DAC: Digital-to-Analog Converter Driver

9.1 Overview

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

The DAC driver includes a basic DAC module (converter) and a DAC buffer.

The basic DAC module supports operations unique to the DAC converter in each DAC instance. The APIs in this part are used in the initialization phase, which enables the DAC module in the application. The APIs enable/disable the clock, enable/disable the module, and configure the converter. Call the initial APIs to prepare the DAC module for the application. The DAC buffer operates the DAC hardware buffer. The DAC module supports a hardware buffer to keep a group of DAC values to be converted. This feature supports updating the DAC output value automatically by triggering the buffer read pointer to move in the buffer. Use the APIs to configure the hardware buffer's trigger mode, watermark, work mode, and use size. Additionally, the APIs operate the DMA, interrupts, flags, the pointer (the index of the buffer), item values, and so on.

Note that the most functional features are designed for the DAC hardware buffer.

9.2 Typical use case

9.2.1 Working as a basic DAC without the hardware buffer feature

```
// ...  
  
// Configures the DAC.  
DAC_GetDefaultConfig(&dacConfigStruct);  
DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);  
DAC_Enable(DEMO_DAC_INSTANCE, true);  
DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U);  
  
// ...  
  
DAC_SetBufferValue(DEMO_DAC_INSTANCE, 0U, dacValue);
```

9.2.2 Working with the hardware buffer

```
// ...  
  
EnableIRQ(DEMO_DAC_IRQ_ID);  
  
// ...  
  
// Configures the DAC.  
DAC_GetDefaultConfig(&dacConfigStruct);  
DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);  
DAC_Enable(DEMO_DAC_INSTANCE, true);
```

Typical use case

```
// Configures the DAC buffer.
DAC_SetDefaultBufferConfig(&dacBufferConfigStruct);
DAC_SetBufferConfig(DEMO_DAC_INSTANCE, &dacBufferConfigStruct);
DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U); // Make sure the read pointer
      to the start.
for (index = 0U, dacValue = 0; index < DEMO_DAC_USED_BUFFER_SIZE; index++, dacValue += (0xFFFFU /
    DEMO_DAC_USED_BUFFER_SIZE))
{
    DAC_SetBufferValue(DEMO_DAC_INSTANCE, index, dacValue);
}
// Clears flags.
#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferWatermarkInterruptFlag = false;
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferReadPointerTopPositionInterruptFlag = false;
    g_DacBufferReadPointerBottomPositionInterruptFlag = false;

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

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

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

Data Structures

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

Enumerations

- enum `_dac_buffer_status_flags` {

 `kDAC_BufferWatermarkFlag` = `DAC_SR_DACBFWMF_MASK`,

 `kDAC_BufferReadPointerTopPositionFlag` = `DAC_SR_DACBFRPTF_MASK`,

 `kDAC_BufferReadPointerBottomPositionFlag` = `DAC_SR_DACBFRPBF_MASK` }

DAC buffer flags.
- enum `_dac_buffer_interrupt_enable` {

 `kDAC_BufferWatermarkInterruptEnable` = `DAC_C0_DACBWIEN_MASK`,

 `kDAC_BufferReadPointerTopInterruptEnable` = `DAC_C0_DACBTIEN_MASK`,

 `kDAC_BufferReadPointerBottomInterruptEnable` = `DAC_C0_DACBBIEN_MASK` }

DAC buffer interrupts.
- enum `dac_reference_voltage_source_t` {

 `kDAC_ReferenceVoltageSourceVref1` = `0U`,

 `kDAC_ReferenceVoltageSourceVref2` = `1U` }

DAC reference voltage source.
- enum `dac_buffer_trigger_mode_t` {

 `kDAC_BufferTriggerByHardwareMode` = `0U`,

 `kDAC_BufferTriggerBySoftwareMode` = `1U` }

DAC buffer trigger mode.
- enum `dac_buffer_watermark_t` {

 `kDAC_BufferWatermark1Word` = `0U`,

 `kDAC_BufferWatermark2Word` = `1U`,

 `kDAC_BufferWatermark3Word` = `2U`,

 `kDAC_BufferWatermark4Word` = `3U` }

DAC buffer watermark.
- enum `dac_buffer_work_mode_t` {

 `kDAC_BufferWorkAsNormalMode` = `0U`,

 `kDAC_BufferWorkAsSwingMode`,

 `kDAC_BufferWorkAsOneTimeScanMode` }

DAC buffer work mode.

Driver version

- `#define FSL_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

DAC driver version 2.0.1.

Initialization

- void `DAC_Init` (`DAC_Type` *base, const `dac_config_t` *config)

Initializes the DAC module.
- void `DAC_Deinit` (`DAC_Type` *base)

De-initializes the DAC module.
- void `DAC_GetDefaultConfig` (`dac_config_t` *config)

Initializes the DAC user configuration structure.
- static void `DAC_Enable` (`DAC_Type` *base, bool enable)

Enables the DAC module.

Data Structure Documentation

Buffer

- static void **DAC_EnableBuffer** (DAC_Type *base, bool enable)
Enables the DAC buffer.
- void **DAC_SetBufferConfig** (DAC_Type *base, const dac_buffer_config_t *config)
Configures the CMP buffer.
- void **DAC_GetDefaultBufferConfig** (dac_buffer_config_t *config)
Initializes the DAC buffer configuration structure.
- static void **DAC_EnableBufferDMA** (DAC_Type *base, bool enable)
Enables the DMA for DAC buffer.
- void **DAC_SetBufferValue** (DAC_Type *base, uint8_t index, uint16_t value)
Sets the value for items in the buffer.
- static void **DAC_DoSoftwareTriggerBuffer** (DAC_Type *base)
Triggers the buffer using software and updates the read pointer of the DAC buffer.
- static uint8_t **DAC_GetBufferReadPointer** (DAC_Type *base)
Gets the current read pointer of the DAC buffer.
- void **DAC_SetBufferReadPointer** (DAC_Type *base, uint8_t index)
Sets the current read pointer of the DAC buffer.
- void **DAC_EnableBufferInterrupts** (DAC_Type *base, uint32_t mask)
Enables interrupts for the DAC buffer.
- void **DAC_DisableBufferInterrupts** (DAC_Type *base, uint32_t mask)
Disables interrupts for the DAC buffer.
- uint32_t **DAC_GetBufferStatusFlags** (DAC_Type *base)
Gets the flags of events for the DAC buffer.
- void **DAC_ClearBufferStatusFlags** (DAC_Type *base, uint32_t mask)
Clears the flags of events for the DAC buffer.

9.3 Data Structure Documentation

9.3.1 struct dac_config_t

Data Fields

- **dac_reference_voltage_source_t referenceVoltageSource**
Select the DAC reference voltage source.
- **bool enableLowPowerMode**
Enable the low-power mode.

9.3.1.0.0.10 Field Documentation

9.3.1.0.0.10.1 **dac_reference_voltage_source_t dac_config_t::referenceVoltageSource**

9.3.1.0.0.10.2 **bool dac_config_t::enableLowPowerMode**

9.3.2 struct dac_buffer_config_t

Data Fields

- **dac_buffer_trigger_mode_t triggerMode**
Select the buffer's trigger mode.

- **dac_buffer_watermark_t watermark**
Select the buffer's watermark.
- **dac_buffer_work_mode_t workMode**
Select the buffer's work mode.
- **uint8_t upperLimit**
Set the upper limit for the buffer index.

9.3.2.0.0.11 Field Documentation

9.3.2.0.0.11.1 dac_buffer_trigger_mode_t dac_buffer_config_t::triggerMode

9.3.2.0.0.11.2 dac_buffer_watermark_t dac_buffer_config_t::watermark

9.3.2.0.0.11.3 dac_buffer_work_mode_t dac_buffer_config_t::workMode

9.3.2.0.0.11.4 uint8_t dac_buffer_config_t::upperLimit

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

9.4 Macro Definition Documentation

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

9.5 Enumeration Type Documentation

9.5.1 enum _dac_buffer_status_flags

Enumerator

kDAC_BufferWatermarkFlag DAC Buffer Watermark Flag.

kDAC_BufferReadPointerTopPositionFlag DAC Buffer Read Pointer Top Position Flag.

kDAC_BufferReadPointerBottomPositionFlag DAC Buffer Read Pointer Bottom Position Flag.

9.5.2 enum _dac_buffer_interrupt_enable

Enumerator

kDAC_BufferWatermarkInterruptEnable DAC Buffer Watermark Interrupt Enable.

kDAC_BufferReadPointerTopInterruptEnable DAC Buffer Read Pointer Top Flag Interrupt Enable.

kDAC_BufferReadPointerBottomInterruptEnable DAC Buffer Read Pointer Bottom Flag Interrupt Enable.

Function Documentation

9.5.3 enum dac_reference_voltage_source_t

Enumerator

kDAC_ReferenceVoltageSourceVref1 The DAC selects DACREF_1 as the reference voltage.

kDAC_ReferenceVoltageSourceVref2 The DAC selects DACREF_2 as the reference voltage.

9.5.4 enum dac_buffer_trigger_mode_t

Enumerator

kDAC_BufferTriggerByHardwareMode The DAC hardware trigger is selected.

kDAC_BufferTriggerBySoftwareMode The DAC software trigger is selected.

9.5.5 enum dac_buffer_watermark_t

Enumerator

kDAC_BufferWatermark1Word 1 word away from the upper limit.

kDAC_BufferWatermark2Word 2 words away from the upper limit.

kDAC_BufferWatermark3Word 3 words away from the upper limit.

kDAC_BufferWatermark4Word 4 words away from the upper limit.

9.5.6 enum dac_buffer_work_mode_t

Enumerator

kDAC_BufferWorkAsNormalMode Normal mode.

kDAC_BufferWorkAsSwingMode Swing mode.

kDAC_BufferWorkAsOneTimeScanMode One-Time Scan mode.

9.6 Function Documentation

9.6.1 void DAC_Init (DAC_Type * *base*, const dac_config_t * *config*)

This function initializes the DAC module including the following operations.

- Enabling the clock for DAC module.
- Configuring the DAC converter with a user configuration.
- Enabling the DAC module.

Parameters

<i>base</i>	DAC peripheral base address.
<i>config</i>	Pointer to the configuration structure. See "dac_config_t".

9.6.2 void DAC_Deinit (DAC_Type * *base*)

This function de-initializes the DAC module including the following operations.

- Disabling the DAC module.
- Disabling the clock for the DAC module.

Parameters

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

9.6.3 void DAC_GetDefaultConfig (dac_config_t * *config*)

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

```
* config->referenceVoltageSource = kDAC_ReferenceVoltageSourceVref2;
* config->enableLowPowerMode = false;
*
```

Parameters

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

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

Parameters

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

Function Documentation

<i>enable</i>	Enables or disables the feature.
---------------	----------------------------------

9.6.5 static void DAC_EnableBuffer (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

9.6.6 void DAC_SetBufferConfig (DAC_Type * *base*, const dac_buffer_config_t * *config*)

Parameters

<i>base</i>	DAC peripheral base address.
<i>config</i>	Pointer to the configuration structure. See "dac_buffer_config_t".

9.6.7 void DAC_GetDefaultBufferConfig (dac_buffer_config_t * *config*)

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

```
* config->triggerMode = kDAC_BufferTriggerBySoftwareMode;
* config->watermark   = kDAC_BufferWatermark1Word;
* config->workMode    = kDAC_BufferWorkAsNormalMode;
* config->upperLimit  = DAC_DATL_COUNT - 1U;
*
```

Parameters

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

9.6.8 static void DAC_EnableBufferDMA (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

9.6.9 void DAC_SetBufferValue (DAC_Type * *base*, uint8_t *index*, uint16_t *value*)

Parameters

<i>base</i>	DAC peripheral base address.
<i>index</i>	Setting the index for items in the buffer. The available index should not exceed the size of the DAC buffer.
<i>value</i>	Setting the value for items in the buffer. 12-bits are available.

9.6.10 static void DAC_DoSoftwareTriggerBuffer (DAC_Type * *base*) [inline], [static]

This function triggers the function using software. The read pointer of the DAC buffer is updated with one step after this function is called. Changing the read pointer depends on the buffer's work mode.

Parameters

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

9.6.11 static uint8_t DAC_GetBufferReadPointer (DAC_Type * *base*) [inline], [static]

This function gets the current read pointer of the DAC buffer. The current output value depends on the item indexed by the read pointer. It is updated either by a software trigger or a hardware trigger.

Parameters

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

Returns

The current read pointer of the DAC buffer.

Function Documentation

9.6.12 void DAC_SetBufferReadPointer (DAC_Type * *base*, uint8_t *index*)

This function sets the current read pointer of the DAC buffer. The current output value depends on the item indexed by the read pointer. It is updated either by a software trigger or a hardware trigger. After the read pointer changes, the DAC output value also changes.

Parameters

<i>base</i>	DAC peripheral base address.
<i>index</i>	Setting an index value for the pointer.

9.6.13 void DAC_EnableBufferInterrupts (DAC_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	DAC peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_dac_buffer_interrupt_enable".

9.6.14 void DAC_DisableBufferInterrupts (DAC_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	DAC peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_dac_buffer_interrupt_enable".

9.6.15 uint32_t DAC_GetBufferStatusFlags (DAC_Type * *base*)

Parameters

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

Returns

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

9.6.16 void DAC_ClearBufferStatusFlags (DAC_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	DAC peripheral base address.
<i>mask</i>	Mask value for flags. See "_dac_buffer_status_flags_t".

Function Documentation

Chapter 10

DMAMUX: Direct Memory Access Multiplexer Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 DMAMUX Operation

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

Driver version

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

DMAMUX Initialization and de-initialization

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

DMAMUX Channel Operation

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

10.3 Macro Definition Documentation

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

Function Documentation

10.4 Function Documentation

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

This function ungates the DMAMUX clock.

Parameters

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

10.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gates the DMAMUX clock.

Parameters

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

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

This function enables the DMAMUX channel.

Parameters

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

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

This function disables the DMAMUX channel.

Note

The user must disable the DMAMUX channel before configuring it.

Parameters

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

Function Documentation

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

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

Parameters

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

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

This function enables the DMAMUX period trigger feature.

Parameters

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

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

This function disables the DMAMUX period trigger.

Parameters

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

Chapter 11

DSPI: Serial Peripheral Interface Driver

11.1 Overview

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

Modules

- DSPI DMA Driver
- DSPI Driver
- DSPI FreeRTOS Driver
- DSPI eDMA Driver
- DSPI µCOS/II Driver
- DSPI µCOS/III 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

```
dspi_master_handle_t g_m_handle; //global variable
dspi_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.enableRx_fifoOverWrite = 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

```
dspi_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 [dsPIC33F DSPI command date configuration used for the SPIx_PUSHR](#). [More...](#)
- struct [dsPIC33F DSPI master ctar configuration structure](#). [More...](#)
- struct [dsPIC33F DSPI master configuration structure](#). [More...](#)
- struct [dsPIC33F DSPI slave ctar configuration structure](#). [More...](#)
- struct [dsPIC33F DSPI slave configuration structure](#). [More...](#)
- struct [dsPIC33F DSPI master/slave transfer structure](#). [More...](#)
- struct [dsPIC33F DSPI master handle structure used for transactional API](#). [More...](#)
- struct [dsPIC33F DSPI slave handle structure used for the transactional API](#). [More...](#)

Macros

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

DSPI Driver

- `#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 {

```

## DSPI Driver

```
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, 3))  
*DSPI driver version 2.1.3.*

## 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, dspi\_master\_handle\_t \*handle, **dspi\_master\_transfer\_callback\_t** callback, void \*userData)  
*Initializes the DSPI master handle.*
- status\_t **DSPI\_MasterTransferBlocking** (SPI\_Type \*base, **dspi\_transfer\_t** \*transfer)  
*DSPI master transfer data using polling.*
- status\_t **DSPI\_MasterTransferNonBlocking** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle, **dspi\_transfer\_t** \*transfer)  
*DSPI master transfer data using interrupts.*
- status\_t **DSPI\_MasterTransferGetCount** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle, size\_t \*count)  
*Gets the master transfer count.*
- void **DSPI\_MasterTransferAbort** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle)  
*DSPI master aborts a transfer using an interrupt.*
- void **DSPI\_MasterTransferHandleIRQ** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle)  
*DSPI Master IRQ handler function.*
- void **DSPI\_SlaveTransferCreateHandle** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle, **dspi\_slave\_transfer\_callback\_t** callback, void \*userData)  
*Initializes the DSPI slave handle.*
- status\_t **DSPI\_SlaveTransferNonBlocking** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle, **dspi\_transfer\_t** \*transfer)  
*DSPI slave transfers data using an interrupt.*
- status\_t **DSPI\_SlaveTransferGetCount** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle, size\_t \*count)  
*Gets the slave transfer count.*
- void **DSPI\_SlaveTransferAbort** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle)  
*DSPI slave aborts a transfer using an interrupt.*
- void **DSPI\_SlaveTransferHandleIRQ** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle)  
*DSPI Master IRQ handler function.*

### 11.2.3 Data Structure Documentation

#### 11.2.3.1 struct dspi\_command\_data\_config\_t

##### Data Fields

- bool **isPcsContinuous**  
*Option to enable the continuous assertion of the chip select between transfers.*
- **dspi\_ctar\_selection\_t whichCtar**  
*The desired Clock and Transfer Attributes Register (CTAR) to use for CTAS.*
- **dspi\_which\_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.*

### 11.2.3.1.0.12 Field Documentation

**11.2.3.1.0.12.1 bool dspi\_command\_data\_config\_t::isPcsContinuous**

**11.2.3.1.0.12.2 dspi\_ctar\_selection\_t dspi\_command\_data\_config\_t::whichCtar**

**11.2.3.1.0.12.3 dspi\_which\_pcs\_t dspi\_command\_data\_config\_t::whichPcs**

**11.2.3.1.0.12.4 bool dspi\_command\_data\_config\_t::isEndOfQueue**

**11.2.3.1.0.12.5 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.*

### 11.2.3.2.0.13 Field Documentation

**11.2.3.2.0.13.1 uint32\_t dspi\_master\_ctar\_config\_t::baudRate**

**11.2.3.2.0.13.2 uint32\_t dspi\_master\_ctar\_config\_t::bitsPerFrame**

**11.2.3.2.0.13.3 dspi\_clock\_polarity\_t dspi\_master\_ctar\_config\_t::cpol**

**11.2.3.2.0.13.4 dspi\_clock\_phase\_t dspi\_master\_ctar\_config\_t::cpha**

**11.2.3.2.0.13.5 dspi\_shift\_direction\_t dspi\_master\_ctar\_config\_t::direction**

**11.2.3.2.0.13.6 uint32\_t dspi\_master\_ctar\_config\_t::pcsToSckDelayInNanoSec**

It also sets the boundary value if out of range.

**11.2.3.2.0.13.7 uint32\_t dspi\_master\_ctar\_config\_t::lastSckToPcsDelayInNanoSec**

It also sets the boundary value if out of range.

**11.2.3.2.0.13.8 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

### 11.2.3.3.0.14 Field Documentation

11.2.3.3.0.14.1 **dspi\_ctar\_selection\_t** **dspi\_master\_config\_t::whichCtar**

11.2.3.3.0.14.2 **dspi\_master\_ctar\_config\_t** **dspi\_master\_config\_t::ctarConfig**

11.2.3.3.0.14.3 **dspi\_which\_pcs\_t** **dspi\_master\_config\_t::whichPcs**

11.2.3.3.0.14.4 **dspi\_pcs\_polarity\_config\_t** **dspi\_master\_config\_t::pcsActiveHighOrLow**

11.2.3.3.0.14.5 **bool** **dspi\_master\_config\_t::enableContinuousSCK**

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

11.2.3.3.0.14.6 **bool** **dspi\_master\_config\_t::enableRx\_fifo\_overWrite**

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.

11.2.3.3.0.14.7 **bool** **dspi\_master\_config\_t::enableModifiedTimingFormat**

11.2.3.3.0.14.8 **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.*

### 11.2.3.4.0.15 Field Documentation

11.2.3.4.0.15.1 **uint32\_t** **dspi\_slave\_ctar\_config\_t::bitsPerFrame**

11.2.3.4.0.15.2 **dspi\_clock\_polarity\_t** **dspi\_slave\_ctar\_config\_t::cpol**

11.2.3.4.0.15.3 **dspi\_clock\_phase\_t** **dspi\_slave\_ctar\_config\_t::cpha**

Slave only supports MSB and does not support LSB.

### 11.2.3.5 struct `dspi_slave_config_t`

#### Data Fields

- `dspi_ctar_selection_t whichCtar`  
*The desired CTAR to use.*
- `dspi_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.*
- `dspi_master_sample_point_t samplePoint`  
*Controls when the module master samples SIN in the Modified Transfer Format.*

#### 11.2.3.5.0.16 Field Documentation

##### 11.2.3.5.0.16.1 `dspi_ctar_selection_t dspi_slave_config_t::whichCtar`

##### 11.2.3.5.0.16.2 `dspi_slave_ctar_config_t dspi_slave_config_t::ctarConfig`

##### 11.2.3.5.0.16.3 `bool dspi_slave_config_t::enableContinuousSCK`

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

##### 11.2.3.5.0.16.4 `bool dspi_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.

##### 11.2.3.5.0.16.5 `bool dspi_slave_config_t::enableModifiedTimingFormat`

##### 11.2.3.5.0.16.6 `dspi_master_sample_point_t dspi_slave_config_t::samplePoint`

It's valid only when CPHA=0.

### 11.2.3.6 struct `dspi_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 \_dspi\_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.

### 11.2.3.6.0.17 Field Documentation

11.2.3.6.0.17.1 `uint8_t* dspi_transfer_t::txData`

11.2.3.6.0.17.2 `uint8_t* dspi_transfer_t::rxData`

11.2.3.6.0.17.3 `volatile size_t dspi_transfer_t::dataSize`

11.2.3.6.0.17.4 `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.*

### 11.2.3.7.0.18 Field Documentation

- 11.2.3.7.0.18.1 `uint32_t dspi_master_handle_t::bitsPerFrame`
- 11.2.3.7.0.18.2 `volatile uint32_t dspi_master_handle_t::command`
- 11.2.3.7.0.18.3 `volatile uint32_t dspi_master_handle_t::lastCommand`
- 11.2.3.7.0.18.4 `uint8_t dspi_master_handle_t::fifoSize`
- 11.2.3.7.0.18.5 `volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer`
- 11.2.3.7.0.18.6 `volatile bool dspi_master_handle_t::isThereExtraByte`
- 11.2.3.7.0.18.7 `uint8_t* volatile dspi_master_handle_t::txData`
- 11.2.3.7.0.18.8 `uint8_t* volatile dspi_master_handle_t::rxData`
- 11.2.3.7.0.18.9 `volatile size_t dspi_master_handle_t::remainingSendByteCount`
- 11.2.3.7.0.18.10 `volatile size_t dspi_master_handle_t::remainingReceiveByteCount`
- 11.2.3.7.0.18.11 `volatile uint8_t dspi_master_handle_t::state`
- 11.2.3.7.0.18.12 `dspi_master_transfer_callback_t dspi_master_handle_t::callback`
- 11.2.3.7.0.18.13 `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.*

### 11.2.3.8.0.19 Field Documentation

11.2.3.8.0.19.1 `uint32_t dspi_slave_handle_t::bitsPerFrame`

11.2.3.8.0.19.2 `volatile bool dspi_slave_handle_t::isThereExtraByte`

11.2.3.8.0.19.3 `uint8_t* volatile dspi_slave_handle_t::txData`

11.2.3.8.0.19.4 `uint8_t* volatile dspi_slave_handle_t::rxData`

11.2.3.8.0.19.5 `volatile size_t dspi_slave_handle_t::remainingSendByteCount`

11.2.3.8.0.19.6 `volatile size_t dspi_slave_handle_t::remainingReceiveByteCount`

11.2.3.8.0.19.7 `volatile uint8_t dspi_slave_handle_t::state`

11.2.3.8.0.19.8 `volatile uint32_t dspi_slave_handle_t::errorCount`

11.2.3.8.0.19.9 `dspi_slave_transfer_callback_t dspi_slave_handle_t::callback`

11.2.3.8.0.19.10 `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, 3))`

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\_S-LAVE 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 \_dsPIC\_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 \_dsPIC\_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 <a href="#">dsPIC33F DSPI API Reference</a> . |
| <i>srcClock_Hz</i>  | Module source input clock in Hertz.                                    |

#### 11.2.7.2 void DSPI\_MasterGetDefaultConfig ( [dsPIC33F DSPI API Reference](#) \* *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:

```

* dsPIC33F_DSPI_MasterConfig_t masterConfig;
* DSPI_MasterGetDefaultConfig(&masterConfig);
*

```

#### Parameters

|                     |                                                          |
|---------------------|----------------------------------------------------------|
| <i>masterConfig</i> | pointer to <a href="#">dsPIC33F DSPI API Reference</a> . |
|---------------------|----------------------------------------------------------|

#### 11.2.7.3 void DSPI\_SlaveInit ( [SPI\\_Type](#) \* *base*, const [dsPIC33F DSPI API Reference](#) \* *slaveConfig* )

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

```

* dsPIC33F_DSPI_SlaveConfig_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_SckToSinc0Clock;
* DSPI_SlaveInit(base, &slaveConfig);
*
```

### Parameters

|                    |                                                                 |
|--------------------|-----------------------------------------------------------------|
| <i>base</i>        | DSPI peripheral address.                                        |
| <i>slaveConfig</i> | Pointer to the structure <a href="#">dspi_master_config_t</a> . |

### 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 <a href="#">dspi_slave_config_t</a> 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 dspi_flags. |

< 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 _dspi_interrupt_enable. |

### 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 _dspi_interrupt_enable. |

### 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 dspi_dma_enable. |

### 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.

```
* SPI_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 PUSHR data register address because this value is needed for the DMA operation.

Parameters

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

Returns

The DSPI master PUSHR data register address.

#### **11.2.7.14 static uint32\_t DSPI\_SlaveGetTxRegisterAddress ( SPI\_Type \* *base* ) [inline], [static]**

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

Parameters

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

Returns

The DSPI slave PUSHR 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 scalar value. This allows the user to directly set the prescaler/scalar 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>dspi_delay_type_t</code> |

### 11.2.7.25 `uint32_t DSPI_MasterSetDelayTimes ( SPI_Type * base, dspi_ctar_selection_t whichCtar, dspi_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 `dspi_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>dspi_ctar_selection_t</code> . |
| <i>whichDelay</i>         | The desired delay to configure, must be of type <code>dspi_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, dspi_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 <b>dspi_command_data_config_t</b> 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 PUSHR 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 PUSHR. 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 PUSHR 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 <a href="#">dspi_transfer_t</a> 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 <a href="#">dspi_master_handle_t</a> structure which stores the transfer state. |
| <i>transfer</i> | Pointer to the <a href="#">dspi_transfer_t</a> 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 <a href="#">dspi_master_handle_t</a> 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 dspi_master_handle_t 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 dspi_master_handle_t 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 dspi_slave_handle_t. |
| <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 <code>dspi_transfer_t</code> 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, dspi\_slave\_dma\_handle\_t \*handle)  
*DSPI slave aborts a transfer which is using DMA.*
- status\_t **DSPI\_SlaveTransferGetCountDMA** (SPI\_Type \*base, dspi\_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 \_dspi\_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 \_dspi\_transfer\_state.*
- **dspi\_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`

### 11.3.2.1.0.20 Field Documentation

11.3.2.1.0.20.1 `uint32_t dspi_master_dma_handle_t::bitsPerFrame`

11.3.2.1.0.20.2 `volatile uint32_t dspi_master_dma_handle_t::command`

11.3.2.1.0.20.3 `volatile uint32_t dspi_master_dma_handle_t::lastCommand`

11.3.2.1.0.20.4 `uint8_t dspi_master_dma_handle_t::fifoSize`

11.3.2.1.0.20.5 `volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer`

11.3.2.1.0.20.6 `volatile bool dspi_master_dma_handle_t::isThereExtraByte`

11.3.2.1.0.20.7 `uint8_t* volatile dspi_master_dma_handle_t::txData`

11.3.2.1.0.20.8 `uint8_t* volatile dspi_master_dma_handle_t::rxData`

11.3.2.1.0.20.9 `volatile size_t dspi_master_dma_handle_t::remainingSendByteCount`

11.3.2.1.0.20.10 `volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount`

11.3.2.1.0.20.11 `uint32_t dspi_master_dma_handle_t::rxBuffIfNull`

11.3.2.1.0.20.12 `uint32_t dspi_master_dma_handle_t::txBuffIfNull`

11.3.2.1.0.20.13 `volatile uint8_t dspi_master_dma_handle_t::state`

11.3.2.1.0.20.14 `dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback`

11.3.2.1.0.20.15 `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.*
- **dspi\_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 \* **dmaTxDataToTxRegHandle**  
*dma\_handle\_t handle point used for TxData to TxReg*

### 11.3.2.2.0.21 Field Documentation

11.3.2.2.0.21.1 `uint32_t dspi_slave_dma_handle_t::bitsPerFrame`

11.3.2.2.0.21.2 `volatile bool dspi_slave_dma_handle_t::isThereExtraByte`

11.3.2.2.0.21.3 `uint8_t* volatile dspi_slave_dma_handle_t::txData`

11.3.2.2.0.21.4 `uint8_t* volatile dspi_slave_dma_handle_t::rxData`

11.3.2.2.0.21.5 `volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount`

11.3.2.2.0.21.6 `volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount`

11.3.2.2.0.21.7 `uint32_t dspi_slave_dma_handle_t::rxBuffIfNull`

11.3.2.2.0.21.8 `uint32_t dspi_slave_dma_handle_t::txBuffIfNull`

11.3.2.2.0.21.9 `uint32_t dspi_slave_dma_handle_t::txLastData`

11.3.2.2.0.21.10 `volatile uint8_t dspi_slave_dma_handle_t::state`

11.3.2.2.0.21.11 `uint32_t dspi_slave_dma_handle_t::errorCount`

11.3.2.2.0.21.12 `dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback`

11.3.2.2.0.21.13 `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*, dsPICMasterDMAHandle\_t \* *handle*, dsPICMasterDMATransferCallback\_t *callback*, void \* *userData*, DMAHandle\_t \* *dmaRxRegToRxDataHandle*, DMAHandle\_t \* *dmaTxDataToIntermediaryHandle*, DMAHandle\_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 <code>dspi_master_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-Intermediary-Handle</i> | <code>dmaTxDataToIntermediaryHandle</code> pointer to <code>dma_handle_t</code> . |
| <i>dma-Intermediary-ToTxReg-Handle</i> | <code>dmaIntermediaryToTxRegHandle</code> pointer to <code>dma_handle_t</code> .  |

#### 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 <code>dspi_master_dma_handle_t</code> structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <code>dspi_transfer_t</code> 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 dspi_master_dma_handle_t 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 dspi_master_dma_handle_t 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 dspi_slave_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-TxRegHandle</i> | dmaTxDataToTxRegHandle pointer to dma_handle_t. |

#### 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 dspi_slave_dma_handle_t structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <a href="#">dspi_transfer_t</a> 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 dspi_slave_dma_handle_t 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 dspi_slave_dma_handle_t 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, dspi\_slave\_edma\_handle\_t \*handle)  
*DSPI slave aborts a transfer which is using eDMA.*
- status\_t **DSPI\_SlaveTransferGetCountEDMA** (SPI\_Type \*base, dspi\_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 \_dspi\_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 , \_dspi\_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.*
- **dspi\_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 dsPICSoftwareTCD [2]`  
`SoftwareTCD`, internal used.

#### 11.4.2.1.0.22 Field Documentation

11.4.2.1.0.22.1 `uint32_t dsPIC_master_edma_handle_t::bitsPerFrame`

11.4.2.1.0.22.2 `volatile uint32_t dsPIC_master_edma_handle_t::command`

11.4.2.1.0.22.3 `volatile uint32_t dsPIC_master_edma_handle_t::lastCommand`

11.4.2.1.0.22.4 `uint8_t dsPIC_master_edma_handle_t::fifoSize`

11.4.2.1.0.22.5 `volatile bool dsPIC_master_edma_handle_t::isPcsActiveAfterTransfer`

11.4.2.1.0.22.6 `uint8_t dsPIC_master_edma_handle_t::nbytes`

11.4.2.1.0.22.7 `volatile uint8_t dsPIC_master_edma_handle_t::state`

11.4.2.1.0.22.8 `uint8_t* volatile dsPIC_master_edma_handle_t::txData`

11.4.2.1.0.22.9 `uint8_t* volatile dsPIC_master_edma_handle_t::rxData`

11.4.2.1.0.22.10 `volatile size_t dsPIC_master_edma_handle_t::remainingSendByteCount`

11.4.2.1.0.22.11 `volatile size_t dsPIC_master_edma_handle_t::remainingReceiveByteCount`

11.4.2.1.0.22.12 `uint32_t dsPIC_master_edma_handle_t::rxBuffIfNull`

11.4.2.1.0.22.13 `uint32_t dsPIC_master_edma_handle_t::txBuffIfNull`

11.4.2.1.0.22.14 `dsPIC_master_edma_transfer_callback_t dsPIC_master_edma_handle_t::callback`

11.4.2.1.0.22.15 `void* dsPIC_master_edma_handle_t::userData`

#### 11.4.2.2 struct \_dsPIC\_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.*
- **dspi\_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*

#### 11.4.2.2.0.23 Field Documentation

- 11.4.2.2.0.23.1 `uint32_t dspi_slave_edma_handle_t::bitsPerFrame`
- 11.4.2.2.0.23.2 `uint8_t* volatile dspi_slave_edma_handle_t::txData`
- 11.4.2.2.0.23.3 `uint8_t* volatile dspi_slave_edma_handle_t::rxData`
- 11.4.2.2.0.23.4 `volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount`
- 11.4.2.2.0.23.5 `volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount`
- 11.4.2.2.0.23.6 `uint32_t dspi_slave_edma_handle_t::rxBuffIfNull`
- 11.4.2.2.0.23.7 `uint32_t dspi_slave_edma_handle_t::txBuffIfNull`
- 11.4.2.2.0.23.8 `uint32_t dspi_slave_edma_handle_t::txLastData`
- 11.4.2.2.0.23.9 `uint8_t dspi_slave_edma_handle_t::nbytes`
- 11.4.2.2.0.23.10 `volatile uint8_t dspi_slave_edma_handle_t::state`
- 11.4.2.2.0.23.11 `dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback`
- 11.4.2.2.0.23.12 `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 <code>edma_handle_t</code> .        |
| <i>edmaTxData-To-Intermediary-Handle</i> | <code>edmaTxDataToIntermediaryHandle</code> pointer to <code>edma_handle_t</code> . |
| <i>edma-Intermediary-ToTxReg-Handle</i>  | <code>edmaIntermediaryToTxRegHandle</code> pointer to <code>edma_handle_t</code> .  |

#### 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 <code>dspi_transfer_t</code> structure.                                           |

Returns

status of `status_t`.

#### 11.4.4.3 `void DSPI_MasterTransferAbortEDMA ( SPI_Type * base, dspi_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 dspi_master_edma_handle_t 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 dspi_master_edma_handle_t 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>  | edmaRxRegToRxDataHandle pointer to <code>edma_handle_t</code> . |
| <i>edmaTxData-ToTxReg-Handle</i> | edmaTxDataToTxRegHandle pointer to <code>edma_handle_t</code> . |

#### 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 <code>dspi_transfer_t</code> 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 dspi_slave_edma_handle_t 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.

## 11.6 DSPI µCOS/II Driver

### 11.6.1 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.*

### 11.6.2 Function Documentation

#### 11.6.2.1 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.

#### 11.6.2.2 status\_t DSPI\_RTOS\_Deinit ( *dspi\_rtos\_handle\_t \* handle* )

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

Parameters

## DSPI µCOS/II Driver

|               |                       |
|---------------|-----------------------|
| <i>handle</i> | The RTOS DSPI handle. |
|---------------|-----------------------|

### 11.6.2.3 **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.

## 11.7 DSPI µCOS/III Driver

### 11.7.1 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.*

### 11.7.2 Function Documentation

#### 11.7.2.1 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.

#### 11.7.2.2 status\_t DSPI\_RTOS\_Deinit ( *dspi\_rtos\_handle\_t \* handle* )

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

Parameters

|               |                       |
|---------------|-----------------------|
| <i>handle</i> | The RTOS DSPI handle. |
|---------------|-----------------------|

### 11.7.2.3 **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 KSDK provides a peripheral driver for the enhanced Direct Memory Access (eDMA) of Kinetis 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_PrepTransfer(&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_GroupPriorityErrorFlag = DMA_ES_GPE_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_QueuemFull = 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_EnableAsyncRequest` (DMA_Type *base, uint32_t channel, bool enable)
Enables an async request for the eDMA transfer.
- static void `EDMA_EnableAutoStopRequest` (DMA_Type *base, uint32_t channel, bool enable)
Enables an auto stop request for the eDMA transfer.
- void `EDMA_EnableChannelInterrupts` (DMA_Type *base, uint32_t channel, uint32_t mask)
Enables the interrupt source for the eDMA transfer.
- void `EDMA_DisableChannelInterrupts` (DMA_Type *base, uint32_t channel, uint32_t mask)
Disables the interrupt source for the eDMA transfer.

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.

Typical use case

- void [EDMA_TcdEnableInterrupts](#) (`edma_tcd_t` *`tcd`, `uint32_t` `mask`)
Enables the interrupt source for the eDMA TCD.
- void [EDMA_TcdDisableInterrupts](#) (`edma_tcd_t` *`tcd`, `uint32_t` `mask`)
Disables the interrupt source for the eDMA TCD.

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_PrepTransfer](#) (`edma_transfer_config_t` *`config`, `void` *`srcAddr`, `uint32_t` `srcWidth`,
`void` *`destAddr`, `uint32_t` `destWidth`, `uint32_t` `bytesEachRequest`, `uint32_t` `transferBytes`, `edma_transfer_type_t` `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.

12.3.1.0.0.24 Field Documentation

12.3.1.0.0.24.1 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.

12.3.1.0.0.24.2 bool edma_config_t::enableHaltOnError

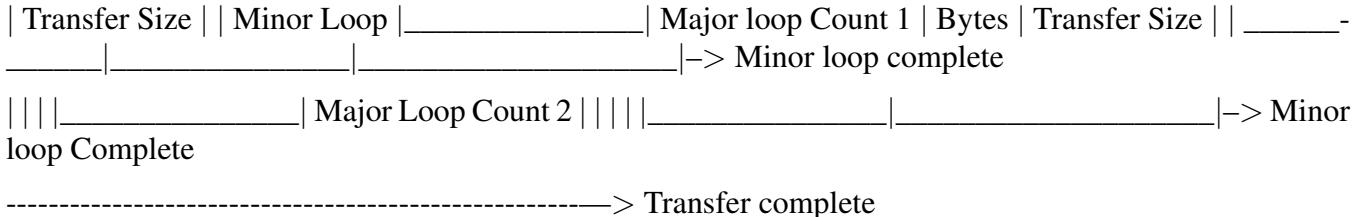
Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

12.3.1.0.0.24.3 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 Structure Documentation

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.

12.3.2.0.0.25 Field Documentation

12.3.2.0.0.25.1 `uint32_t edma_transfer_config_t::srcAddr`

12.3.2.0.0.25.2 `uint32_t edma_transfer_config_t::destAddr`

12.3.2.0.0.25.3 `edma_transfer_size_t edma_transfer_config_t::srcTransferSize`

12.3.2.0.0.25.4 `edma_transfer_size_t edma_transfer_config_t::destTransferSize`

12.3.2.0.0.25.5 `int16_t edma_transfer_config_t::srcOffset`

12.3.2.0.0.25.6 `int16_t edma_transfer_config_t::destOffset`

12.3.2.0.0.25.7 `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.

12.3.4.0.0.26 Field Documentation

12.3.4.0.0.26.1 bool `edma_minor_offset_config_t::enableSrcMinorOffset`

12.3.4.0.0.26.2 bool `edma_minor_offset_config_t::enableDestMinorOffset`

12.3.4.0.0.26.3 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

12.3.5.0.0.27 Field Documentation

12.3.5.0.0.27.1 `__IO uint16_t edma_tcd_t::CITER`

12.3.5.0.0.27.2 `__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.

12.3.6.0.0.28 Field Documentation

12.3.6.0.0.28.1 `edma_callback edma_handle_t::callback`

12.3.6.0.0.28.2 `void* edma_handle_t::userData`

12.3.6.0.0.28.3 `DMA_Type* edma_handle_t::base`

12.3.6.0.0.28.4 `edma_tcd_t* edma_handle_t::tcdPool`

12.3.6.0.0.28.5 `uint8_t edma_handle_t::channel`

12.3.6.0.0.28.6 `volatile int8_t edma_handle_t::header`

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

12.3.6.0.0.28.7 `volatile int8_t edma_handle_t::tail`

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

12.3.6.0.0.28.8 volatile int8_t edma_handle_t::tcdUsed

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

12.3.6.0.0.28.9 volatile int8_t edma_handle_t::tcdSize**12.3.6.0.0.28.10 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 tclds)****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

<i>kEDMA_Modulo2Kbytes</i>	Circular buffer size is 2 K bytes.
<i>kEDMA_Modulo4Kbytes</i>	Circular buffer size is 4 K bytes.
<i>kEDMA_Modulo8Kbytes</i>	Circular buffer size is 8 K bytes.
<i>kEDMA_Modulo16Kbytes</i>	Circular buffer size is 16 K bytes.
<i>kEDMA_Modulo32Kbytes</i>	Circular buffer size is 32 K bytes.
<i>kEDMA_Modulo64Kbytes</i>	Circular buffer size is 64 K bytes.
<i>kEDMA_Modulo128Kbytes</i>	Circular buffer size is 128 K bytes.
<i>kEDMA_Modulo256Kbytes</i>	Circular buffer size is 256 K bytes.
<i>kEDMA_Modulo512Kbytes</i>	Circular buffer size is 512 K bytes.
<i>kEDMA_Modulo1Mbytes</i>	Circular buffer size is 1 M bytes.
<i>kEDMA_Modulo2Mbytes</i>	Circular buffer size is 2 M bytes.
<i>kEDMA_Modulo4Mbytes</i>	Circular buffer size is 4 M bytes.
<i>kEDMA_Modulo8Mbytes</i>	Circular buffer size is 8 M bytes.
<i>kEDMA_Modulo16Mbytes</i>	Circular buffer size is 16 M bytes.
<i>kEDMA_Modulo32Mbytes</i>	Circular buffer size is 32 M bytes.
<i>kEDMA_Modulo64Mbytes</i>	Circular buffer size is 64 M bytes.
<i>kEDMA_Modulo128Mbytes</i>	Circular buffer size is 128 M bytes.
<i>kEDMA_Modulo256Mbytes</i>	Circular buffer size is 256 M bytes.
<i>kEDMA_Modulo512Mbytes</i>	Circular buffer size is 512 M bytes.
<i>kEDMA_Modulo1Gbytes</i>	Circular buffer size is 1 G bytes.
<i>kEDMA_Modulo2Gbytes</i>	Circular buffer size is 2 G bytes.

12.6.3 enum edma_bandwidth_t

Enumerator

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

12.6.4 enum edma_channel_link_type_t

Enumerator

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

12.6.5 enum _edma_channel_status_flags

Enumerator

<i>kEDMA_DoneFlag</i>	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_GroupPriorityErrorFlag Group priority is not unique.

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.

Function Documentation

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.

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, &tcd);
*
```

Parameters

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

Function Documentation

<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.

Function Documentation

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
<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_EnableAsyncRequest (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

Parameters

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

12.7.12 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).

12.7.13 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.

Function Documentation

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

Parameters

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

12.7.15 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.

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

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The 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.

Function Documentation

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.

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

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

Parameters

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

12.7.18 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.

12.7.19 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

12.7.20 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.

12.7.21 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).

Function Documentation

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

Parameters

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

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

Parameters

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

12.7.24 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.

12.7.25 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.

Function Documentation

12.7.26 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.

12.7.27 **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)

12.7.28 **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.

12.7.29 **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.

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

Parameters

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

12.7.31 **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.

12.7.32 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.

12.7.33 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.

12.7.34 void EDMA_PreparesTransfer(edma_transfer_config_t * *config*, void * *srcAddr*, uint32_t *srcWidth*, void * *destAddr*, uint32_t *destWidth*, uint32_t *bytesEachRequest*, uint32_t *transferBytes*, edma_transfer_type_t *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 edma_transfer_t.
<i>srcAddr</i>	eDMA transfer source address.
<i>srcWidth</i>	eDMA transfer source address width(bytes).
<i>destAddr</i>	eDMA transfer destination address.
<i>destWidth</i>	eDMA transfer destination address width(bytes).
<i>bytesEachRequest</i>	eDMA transfer bytes per channel request.
<i>transferBytes</i>	eDMA transfer bytes to be transferred.
<i>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).

12.7.35 **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.
--------------------------	---

12.7.36 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.
---------------	----------------------

12.7.37 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.
---------------	----------------------

12.7.38 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.
---------------	---------------------

12.7.39 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

ENET: Ethernet MAC Driver

13.1 Overview

The KSDK provides a peripheral driver for the 10/100 Mbps Ethernet MAC (ENET) module of Kinetis devices.

The MII interface is the interface connected with MAC and PHY. the Serial management interface - MII management interface should be set before any access to the external PHY chip register. Call [ENET_SetSMI\(\)](#) to initialize MII management interface. Use [ENET_StartSMIRead\(\)](#), [ENET_StartSMIWrite\(\)](#), and [ENET_ReadSMIData\(\)](#) to read/write to PHY registers. This function group sets up the MII and serial management SMI interface, gets data from the SMI interface, and starts the SMI read and write command. Use [ENET_SetMII\(\)](#) to configure the MII before successfully getting data from the external PHY.

This group sets/gets the ENET mac address and the multicast group address filter. [ENET_AddMulticast-Group\(\)](#) should be called to add the ENET MAC to the multicast group. The IEEE 1588 feature requires receiving the PTP message.

For ENET receive, the [ENET_GetRxFrameSize\(\)](#) function must be called to get the received data size. Then, call the [ENET_ReadFrame\(\)](#) function to get the received data. If the received error occurs, call the [ENET_GetRxErrBeforeReadFrame\(\)](#) function after [ENET_GetRxFrameSize\(\)](#) and before [ENET_ReadFrame\(\)](#) functions to get the detailed error information.

For ENET transmit, call the [ENET_SendFrame\(\)](#) function to send the data out. The transmit data error information is only accessible for the IEEE 1588 enhanced buffer descriptor mode. When the [ENET_ENHANCEDBUFFERDESCRIPTOR_MODE](#) is defined, the [ENET_GetTxErrAfterSendFrame\(\)](#) can be used to get the detail transmit error information. The transmit error information can only be updated by uDMA after the data is transmitted. The [ENET_GetTxErrAfterSendFrame\(\)](#) function is recommended to be called on the transmit interrupt handler.

This function group configures the PTP IEEE 1588 feature, starts/stops/gets/sets/adjusts the PTP IEEE 1588 timer, gets the receive/transmit frame timestamp, and PTP IEEE 1588 timer channel feature setting.

The [ENET_Ptp1588Configure\(\)](#) function must be called when the [ENET_ENHANCEDBUFFERDESCRIPTOR_MODE](#) is defined and the IEEE 1588 feature is required. The [ENET_GetRxFrameTime\(\)](#) and [ENET_GetTxFrameTime\(\)](#) functions are called by the PTP stack to get the timestamp captured by the ENET driver.

13.2 Typical use case

13.2.1 ENET Initialization, receive, and transmit operations

For the [ENET_ENHANCEDBUFFERDESCRIPTOR_MODE](#) undefined use case, use the legacy type buffer descriptor transmit/receive the frame as follows.

Typical use case

```
enet_config_t config;
uint32_t length = 0;
uint32_t sysClock;
uint32_t phyAddr = 0;
bool link = false;
phy_speed_t speed;
phy_duplex_t duplex;
enet_status_t result;
enet_data_error_stats_t eErrorStatic;
// Prepares the buffer configuration.
enet_buffer_config_t buffCfg =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
    ENET_BuffSizeAlign(ENET_TXBUFF_SIZE),
    &RxBuffDesrip[0], // Prepare buffers
    &TxBuffDesrip[0], // Prepare buffers
    &RxDataBuff[0][0], // Prepare buffers
    &TxDataBuff[0][0], // Prepare buffers
};

sysClock = CLOCK_GetFreq(kCLOCK_CoreSysClk);

// Gets the default configuration.
ENET_GetDefaultConfig(&config);
PHY_Init(EXAMPLE_ENET, 0, sysClock);
// Changes the link status to PHY auto-negotiated link status.
PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link);
if (link)
{
    PHY_GetLinkSpeedDuplex(EXAMPLE_ENET, phyAddr, &speed, &duplex);
    config.miSpeed = (enet_mi_speed_t)speed;
    config.miDuplex = (enet_mi_duplex_t)duplex;
}
ENET_Init(EXAMPLE_ENET, &handle, &config, &buffCfg, &macAddr[0], sysClock);
ENET_ActiveRead(EXAMPLE_ENET);

while (1)
{
    // Gets the frame size.
    result = ENET_GetRxFrameSize(&handle, &length);
    // Calls the ENET_ReadFrame when there is a received frame.
    if (length != 0)
    {
        // Receives a valid frame and delivers the receive buffer with the size equal to length.
        uint8_t *data = (uint8_t *)malloc(length);
        ENET_ReadFrame(EXAMPLE_ENET, &handle, data, length);
        // Delivers the data to the upper layer.
        .....
        free(data);
    }
    else if (result == kStatus_ENET_RxFrameErr)
    {
        // Updates the received buffer when an error occurs.
        ENET_GetRxErrBeforeReadFrame(&handle, &eErrStatic);
        // Updates the receive buffer.
        ENET_ReadFrame(EXAMPLE_ENET, &handle, NULL, 0);
    }

    // Sends a multicast frame when the PHY is linked up.
    if (kStatus_Success == PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link))
    {
        if (link)
        {
            ENET_SendFrame(EXAMPLE_ENET, &handle, &frame[0], ENET_DATA_LENGTH);
        }
    }
}
```

}

For the ENET_ENHANCEDBUFFERDESCRIPTOR_MODE defined use case, add the PTP IEEE 1588 configuration to enable the PTP IEEE 1588 feature. The initialization occurs as follows.

```

enet_config_t config;
uint32_t length = 0;
uint32_t sysClock;
uint32_t phyAddr = 0;
bool link = false;
phy_speed_t speed;
phy_duplex_t duplex;
enet_status_t result;
enet_data_err_stats_t eErrStatic;
enet_buffer_config_t buffCfg =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    ENET_BuffSizeAlign(ENET_RXBUFF_SIZE),
    ENET_BuffSizeAlign(ENET_TXBUFF_SIZE),
    &RxBuffDescrip[0],
    &TxBuffDescrip[0],
    &RxDataBuff[0][0],
    &TxDataBuff[0][0],
};

sysClock = CLOCK_GetFreq(kCLOCK_CoreSysClk);

// Sets the PTP 1588 source.
CLOCK_SetEnetTime0Clock(2);
ptpClock = CLOCK_GetFreq(kCLOCK_Osc0ErClk);
// Prepares the PTP configuration.
enet_ptp_config_t ptpConfig =
{
    ENET_RXBD_NUM,
    ENET_TXBD_NUM,
    &g_rxPtpTsBuff[0],
    &g_txPtpTsBuff[0],
    kENET_PtpTimerChannel1,
    ptpClock,
};

// Gets the default configuration.
ENET_GetDefaultConfig(&config);

PHY_Init(EXAMPLE_ENET, 0, sysClock);
// Changes the link status to PHY auto-negotiated link status.
PHY_GetLinkStatus(EXAMPLE_ENET, phyAddr, &link);
if (link)
{
    PHY_GetLinkSpeedDuplex(EXAMPLE_ENET, phyAddr, &speed, &duplex);
    config.miiSpeed = (enet_mii_speed_t)speed;
    config.miiDuplex = (enet_mii_duplex_t)duplex;
}

ENET_Init(EXAMPLE_ENET, &handle, &config, &buffCfg, &macAddr[0], sysClock);

// Configures the PTP 1588 feature.
ENET_Ptp1588Configure(EXAMPLE_ENET, &handle, &ptpConfig);
// Adds the device to the PTP multicast group.
ENET_AddMulticastGroup(EXAMPLE_ENET, &mGAddr[0]);

ENET_ActiveRead(EXAMPLE_ENET);

```

Typical use case

Data Structures

- struct [enet_rx_bd_struct_t](#)
Defines the receive buffer descriptor structure for the little endian system. [More...](#)
- struct [enet_tx_bd_struct_t](#)
Defines the enhanced transmit buffer descriptor structure for the little endian system. [More...](#)
- struct [enet_data_error_stats_t](#)
Defines the ENET data error statistic structure. [More...](#)
- struct [enet_buffer_config_t](#)
Defines the receive buffer descriptor configuration structure. [More...](#)
- struct [enet_config_t](#)
Defines the basic configuration structure for the ENET device. [More...](#)
- struct [enet_handle_t](#)
Defines the ENET handler structure. [More...](#)

Macros

- #define [ENET_BUFFDESCRIPTOR_RX_ERR_MASK](#)
Defines the receive error status flag mask.
- #define [ENET_FIFO_MIN_RX_FULL](#) 5U
ENET minimum receive FIFO full.
- #define [ENET_RX_MIN_BUFFERSIZE](#) 256U
ENET minimum buffer size.
- #define [ENET_BUFF_ALIGNMENT](#) 16U
Ethernet buffer alignment.
- #define [ENET_PHY_MAXADDRESS](#) (ENET_MMFR_PA_MASK >> ENET_MMFR_PA_SHIFT)
Defines the PHY address scope for the ENET.

Typedefs

- typedef void(* [enet_callback_t](#)) (ENET_Type *base, enet_handle_t *handle, [enet_event_t](#) event, void *userData)
ENET callback function.

Enumerations

- enum [_enet_status](#) {
 kStatus_ENET_RxFrameError = MAKE_STATUS(kStatusGroup_ENET, 0U),
 kStatus_ENET_RxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 1U),
 kStatus_ENET_RxFrameEmpty = MAKE_STATUS(kStatusGroup_ENET, 2U),
 kStatus_ENET_TxFrameBusy,
 kStatus_ENET_TxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 4U) }
Defines the status return codes for transaction.
- enum [enet_mii_mode_t](#) {
 kENET_MiiMode = 0U,
 kENET_RmiiMode }
Defines the RMII or MII mode for data interface between the MAC and the PHY.

- enum `enet_mii_speed_t` {

 `kENET_MiiSpeed10M` = 0U,

 `kENET_MiiSpeed100M` }

 Defines the 10 Mbps or 100 Mbps speed for the MII data interface.
- enum `enet_mii_duplex_t` {

 `kENET_MiiHalfDuplex` = 0U,

 `kENET_MiiFullDuplex` }

 Defines the half or full duplex for the MII data interface.
- enum `enet_mii_write_t` {

 `kENET_MiiWriteNoCompliant` = 0U,

 `kENET_MiiWriteValidFrame` }

 Defines the write operation for the MII management frame.
- enum `enet_mii_read_t` {

 `kENET_MiiReadValidFrame` = 2U,

 `kENET_MiiReadNoCompliant` = 3U }

 Defines the read operation for the MII management frame.
- enum `enet_special_control_flag_t` {

 `kENET_ControlFlowControlEnable` = 0x0001U,

 `kENET_ControlRxPayloadCheckEnable` = 0x0002U,

 `kENET_ControlRxPadRemoveEnable` = 0x0004U,

 `kENET_ControlRxBroadCastRejectEnable` = 0x0008U,

 `kENET_ControlMacAddrInsert` = 0x0010U,

 `kENET_ControlStoreAndFwdDisable` = 0x0020U,

 `kENET_ControlSMIPreambleDisable` = 0x0040U,

 `kENET_ControlPromiscuousEnable` = 0x0080U,

 `kENET_ControlMIILoopEnable` = 0x0100U,

 `kENET_ControlVLANTagEnable` = 0x0200U }

 Defines a special configuration for ENET MAC controller.
- enum `enet_interrupt_enable_t` {

 `kENET_BabrInterrupt` = ENET_EIR_BABR_MASK,

 `kENET_BabtInterrupt` = ENET_EIR_BABT_MASK,

 `kENET_GraceStopInterrupt` = ENET_EIR_GRA_MASK,

 `kENET_TxFrameInterrupt` = ENET_EIR_TXF_MASK,

 `kENET_TxBufferInterrupt` = ENET_EIR_TXB_MASK,

 `kENET_RxFrameInterrupt` = ENET_EIR_RXF_MASK,

 `kENET_RxBufferInterrupt` = ENET_EIR_RXB_MASK,

 `kENET_MiiInterrupt` = ENET_EIR_MII_MASK,

 `kENET_EBusERInterrupt` = ENET_EIR_EBERR_MASK,

 `kENET_LateCollisionInterrupt` = ENET_EIR_LC_MASK,

 `kENET_RetryLimitInterrupt` = ENET_EIR_RL_MASK,

 `kENET_UnderrunInterrupt` = ENET_EIR_UN_MASK,

 `kENET_PayloadRxInterrupt` = ENET_EIR_PLR_MASK,

 `kENET_WakeupInterrupt` = ENET_EIR_WAKEUP_MASK }

 List of interrupts supported by the peripheral.
- enum `enet_event_t` {

Typical use case

```
kENET_RxEvent,  
kENET_TxEvent,  
kENET_ErrEvent,  
kENET_WakeUpEvent }
```

Defines the common interrupt event for callback use.

- enum `enet_tx_accelerator_t`{
kENET_TxAccelIsShift16Enabled = ENET_TACC_SHIFT16_MASK,
kENET_TxAccelIpCheckEnabled = ENET_TACC_IPCHK_MASK,
kENET_TxAccelProtoCheckEnabled = ENET_TACC_PROCHK_MASK }

Defines the transmit accelerator configuration.

- enum `enet_rx_accelerator_t`{
kENET_RxAccelPadRemoveEnabled = ENET_RACC_PADREM_MASK,
kENET_RxAccelIpCheckEnabled = ENET_RACC_IPDIS_MASK,
kENET_RxAccelProtoCheckEnabled = ENET_RACC_PRODIS_MASK,
kENET_RxAccelMacCheckEnabled = ENET_RACC_LINEDIS_MASK,
kENET_RxAccelIsShift16Enabled = ENET_RACC_SHIFT16_MASK }

Defines the receive accelerator configuration.

Driver version

- #define `FSL_ENET_DRIVER_VERSION` (MAKE_VERSION(2, 0, 1))
Defines the driver version.

Control and status region bit masks of the receive buffer descriptor.

- #define `ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK` 0x8000U
Empty bit mask.
- #define `ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK` 0x4000U
Software owner one mask.
- #define `ENET_BUFFDESCRIPTOR_RX_WRAP_MASK` 0x2000U
Next buffer descriptor is the start address.
- #define `ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask` 0x1000U
Software owner two mask.
- #define `ENET_BUFFDESCRIPTOR_RX_LAST_MASK` 0x0800U
Last BD of the frame mask.
- #define `ENET_BUFFDESCRIPTOR_RX_MISS_MASK` 0x0100U
Received because of the promiscuous mode.
- #define `ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK` 0x0080U
Broadcast packet mask.
- #define `ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK` 0x0040U
Multicast packet mask.
- #define `ENET_BUFFDESCRIPTOR_RX_LENVIOLATE_MASK` 0x0020U
Length violation mask.
- #define `ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK` 0x0010U
Non-octet aligned frame mask.
- #define `ENET_BUFFDESCRIPTOR_RX_CRC_MASK` 0x0004U
CRC error mask.
- #define `ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK` 0x0002U
FIFO overrun mask.

- #define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U
Frame is truncated mask.

Control and status bit masks of the transmit buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U
Ready bit mask.
- #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U
Software owner one mask.
- #define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U
Wrap buffer descriptor mask.
- #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U
Software owner two mask.
- #define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U
Last BD of the frame mask.
- #define ENET_BUFFDESCRIPTOR_TX_TRANMICRC_MASK 0x0400U
Transmit CRC mask.

Defines the maximum Ethernet frame size.

- #define ENET_FRAME_MAX_FRAMELEN 1518U
Maximum Ethernet frame size.
- #define ENET_FRAME_MAX_VALNFRAMELEN 1522U
Maximum VLAN frame size.

Initialization and de-initialization

- void ENET_GetDefaultConfig (enet_config_t *config)
Gets the ENET default configuration structure.
- void ENET_Init (ENET_Type *base, enet_handle_t *handle, const enet_config_t *config, const enet_buffer_config_t *bufferConfig, uint8_t *macAddr, uint32_t srcClock_Hz)
Initializes the ENET module.
- void ENET_Deinit (ENET_Type *base)
Deinitializes the ENET module.
- static void ENET_Reset (ENET_Type *base)
Resets the ENET module.

MII interface operation

- void ENET_SetMII (ENET_Type *base, enet_mii_speed_t speed, enet_mii_duplex_t duplex)
Sets the ENET MII speed and duplex.
- void ENET_SetSMI (ENET_Type *base, uint32_t srcClock_Hz, bool isPreambleDisabled)
Sets the ENET SMI (serial management interface) - MII management interface.
- static bool ENET_GetSMI (ENET_Type *base)
Gets the ENET SMI- MII management interface configuration.
- static uint32_t ENET_ReadSMIData (ENET_Type *base)
Reads data from the PHY register through an SMI interface.
- void ENET_StartSMIRead (ENET_Type *base, uint32_t phyAddr, uint32_t phyReg, enet_mii_read_t operation)
Starts an SMI (Serial Management Interface) read command.

Typical use case

- void [ENET_StartSMIWrite](#) (ENET_Type *base, uint32_t phyAddr, uint32_t phyReg, [enet_mii_write_t](#) operation, uint32_t data)
Starts an SMI write command.

MAC Address Filter

- void [ENET_SetMacAddr](#) (ENET_Type *base, uint8_t *macAddr)
Sets the ENET module Mac address.
- void [ENET_GetMacAddr](#) (ENET_Type *base, uint8_t *macAddr)
Gets the ENET module Mac address.
- void [ENET_AddMulticastGroup](#) (ENET_Type *base, uint8_t *address)
Adds the ENET device to a multicast group.
- void [ENET_LeaveMulticastGroup](#) (ENET_Type *base, uint8_t *address)
Moves the ENET device from a multicast group.

Other basic operations

- static void [ENET_ActiveRead](#) (ENET_Type *base)
Activates ENET read or receive.
- static void [ENET_EnableSleepMode](#) (ENET_Type *base, bool enable)
Enables/disables the MAC to enter sleep mode.
- static void [ENET_GetAccelFunction](#) (ENET_Type *base, uint32_t *txAccelOption, uint32_t *rxAccelOption)
Gets ENET transmit and receive accelerator functions from the MAC controller.

Interrupts

- static void [ENET_EnableInterrupts](#) (ENET_Type *base, uint32_t mask)
Enables the ENET interrupt.
- static void [ENET_DisableInterrupts](#) (ENET_Type *base, uint32_t mask)
Disables the ENET interrupt.
- static uint32_t [ENET_GetInterruptStatus](#) (ENET_Type *base)
Gets the ENET interrupt status flag.
- static void [ENET_ClearInterruptStatus](#) (ENET_Type *base, uint32_t mask)
Clears the ENET interrupt events status flag.

Transactional operation

- void [ENET_SetCallback](#) (enet_handle_t *handle, [enet_callback_t](#) callback, void *userData)
Sets the callback function.
- void [ENET_GetRxErrBeforeReadFrame](#) (enet_handle_t *handle, [enet_data_error_stats_t](#) *eErrorStatic)
Gets the ENET the error statistics of a received frame.
- status_t [ENET_GetRxFrameSize](#) (enet_handle_t *handle, uint32_t *length)
Gets the size of the read frame.
- status_t [ENET_ReadFrame](#) (ENET_Type *base, enet_handle_t *handle, uint8_t *data, uint32_t length)
Reads a frame from the ENET device.
- status_t [ENET_SendFrame](#) (ENET_Type *base, enet_handle_t *handle, uint8_t *data, uint32_t length)

- `void ENET_TransmitIRQHandler (ENET_Type *base, enet_handle_t *handle)`
The transmit IRQ handler.
- `void ENET_ReceiveIRQHandler (ENET_Type *base, enet_handle_t *handle)`
The receive IRQ handler.
- `void ENET_ErrorIRQHandler (ENET_Type *base, enet_handle_t *handle)`
The error IRQ handler.

13.3 Data Structure Documentation

13.3.1 struct enet_rx_bd_struct_t

Data Fields

- `uint16_t length`
Buffer descriptor data length.
- `uint16_t control`
Buffer descriptor control and status.
- `uint8_t * buffer`
Data buffer pointer.

13.3.1.0.0.29 Field Documentation

13.3.1.0.0.29.1 `uint16_t enet_rx_bd_struct_t::length`

13.3.1.0.0.29.2 `uint16_t enet_rx_bd_struct_t::control`

13.3.1.0.0.29.3 `uint8_t* enet_rx_bd_struct_t::buffer`

13.3.2 struct enet_tx_bd_struct_t

Data Fields

- `uint16_t length`
Buffer descriptor data length.
- `uint16_t control`
Buffer descriptor control and status.
- `uint8_t * buffer`
Data buffer pointer.

Data Structure Documentation

13.3.2.0.0.30 Field Documentation

13.3.2.0.0.30.1 `uint16_t enet_tx_bd_struct_t::length`

13.3.2.0.0.30.2 `uint16_t enet_tx_bd_struct_t::control`

13.3.2.0.0.30.3 `uint8_t* enet_tx_bd_struct_t::buffer`

13.3.3 `struct enet_data_error_stats_t`

Data Fields

- `uint32_t statsRxLenGreaterErr`
Receive length greater than RCR[MAX_FL].
- `uint32_t statsRxAlignErr`
Receive non-octet alignment.
- `uint32_t statsRxFcsErr`
Receive CRC error.
- `uint32_t statsRxOverRunErr`
Receive over run.
- `uint32_t statsRxTruncateErr`
Receive truncate.

13.3.3.0.0.31 Field Documentation

13.3.3.0.0.31.1 `uint32_t enet_data_error_stats_t::statsRxLenGreaterErr`

13.3.3.0.0.31.2 `uint32_t enet_data_error_stats_t::statsRxFcsErr`

13.3.3.0.0.31.3 `uint32_t enet_data_error_stats_t::statsRxOverRunErr`

13.3.3.0.0.31.4 `uint32_t enet_data_error_stats_t::statsRxTruncateErr`

13.3.4 `struct enet_buffer_config_t`

Note that for the internal DMA requirements, the buffers have a corresponding alignment requirements.

1. The aligned receive and transmit buffer size must be evenly divisible by 16.
2. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned.
However, it's recommended to be evenly divisible by 16.
3. The aligned transmit and receive buffer start address must be evenly divisible by 16. Receive buffers should be continuous with the total size equal to "rxBdNumber * rxBuffSizeAlign". Transmit buffers should be continuous with the total size equal to "txBdNumber * txBuffSizeAlign".

Data Fields

- `uint16_t rxBdNumber`
Receive buffer descriptor number.

- `uint16_t txBdNumber`
Transmit buffer descriptor number.
- `uint32_t rxBuffSizeAlign`
Aligned receive data buffer size.
- `uint32_t txBuffSizeAlign`
Aligned transmit data buffer size.
- `volatile enet_rx_bd_struct_t * rxBdStartAddrAlign`
Aligned receive buffer descriptor start address.
- `volatile enet_tx_bd_struct_t * txBdStartAddrAlign`
Aligned transmit buffer descriptor start address.
- `uint8_t * rxBufferAlign`
Receive data buffer start address.
- `uint8_t * txBufferAlign`
Transmit data buffer start address.

13.3.4.0.0.32 Field Documentation

13.3.4.0.0.32.1 `uint16_t enet_buffer_config_t::rxBdNumber`

13.3.4.0.0.32.2 `uint16_t enet_buffer_config_t::txBdNumber`

13.3.4.0.0.32.3 `uint32_t enet_buffer_config_t::rxBuffSizeAlign`

13.3.4.0.0.32.4 `uint32_t enet_buffer_config_t::txBuffSizeAlign`

13.3.4.0.0.32.5 `volatile enet_rx_bd_struct_t* enet_buffer_config_t::rxBdStartAddrAlign`

13.3.4.0.0.32.6 `volatile enet_tx_bd_struct_t* enet_buffer_config_t::txBdStartAddrAlign`

13.3.4.0.0.32.7 `uint8_t* enet_buffer_config_t::rxBufferAlign`

13.3.4.0.0.32.8 `uint8_t* enet_buffer_config_t::txBufferAlign`

13.3.5 `struct enet_config_t`

Note:

1. `macSpecialConfig` is used for a special control configuration, a logical OR of "enet_special_control_flag_t". For a special configuration for MAC, set this parameter to 0.
2. `txWatermark` is used for a cut-through operation. It is in steps of 64 bytes. 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO 3 - 192 bytes written to TX FIFO The maximum of `txWatermark` is 0x2F - 4032 bytes written to TX FIFO. `txWatermark` allows minimizing the transmit latency to set the `txWatermark` to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
3. `rxFifoFullThreshold` is similar to the `txWatermark` for cut-through operation in RX. It is in 64-bit words. The minimum is `ENET_FIFO_MIN_RX_FULL` and the maximum is `0xFF`. If the end of the frame is stored in FIFO and the frame size if smaller than the `txWatermark`, the frame is still transmitted. The rule is the same for `rxFifoFullThreshold` in the receive direction.
4. When "kENET_ControlFlowControlEnable" is set in the `macSpecialConfig`, ensure that the pause-

Data Structure Documentation

- Duration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.
5. When "kENET_ControlStoreAndFwdDisabled" is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
 6. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The "enet_tx_accelerator_t" and "enet_rx_accelerator_t" are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.

Data Fields

- **uint32_t macSpecialConfig**
Mac special configuration.
- **uint32_t interrupt**
Mac interrupt source.
- **uint16_t rxMaxFrameLen**
Receive maximum frame length.
- **enet_mii_mode_t miiMode**
MII mode.
- **enet_mii_speed_t miiSpeed**
MII Speed.
- **enet_mii_duplex_t miiDuplex**
MII duplex.
- **uint8_t rxAccelerConfig**
Receive accelerator, A logical OR of "enet_rx_accelerator_t".
- **uint8_t txAccelerConfig**
Transmit accelerator, A logical OR of "enet_rx_accelerator_t".
- **uint16_t pauseDuration**
For flow control enabled case: Pause duration.
- **uint8_t rxFifoEmptyThreshold**
For flow control enabled case: when RX FIFO level reaches this value, it makes MAC generate XOFF pause frame.
- **uint8_t rxFifoStatEmptyThreshold**
For flow control enabled case: number of frames in the receive FIFO, independent of size, that can be accept.
- **uint8_t rxFifoFullThreshold**
For store and forward disable case, the data required in RX FIFO to notify the MAC receive ready status.
- **uint8_t txFifoWatermark**
For store and forward disable case, the data required in TX FIFO before a frame transmit start.

13.3.5.0.0.33 Field Documentation

13.3.5.0.0.33.1 uint32_t enet_config_t::macSpecialConfig

A logical OR of "enet_special_control_flag_t".

13.3.5.0.0.33.2 uint32_t enet_config_t::interrupt

A logical OR of "enet_interrupt_enable_t".

13.3.5.0.0.33.3 uint16_t enet_config_t::rxMaxFrameLen**13.3.5.0.0.33.4 enet_mii_mode_t enet_config_t::miiMode****13.3.5.0.0.33.5 enet_mii_speed_t enet_config_t::miiSpeed****13.3.5.0.0.33.6 enet_mii_duplex_t enet_config_t::miiDuplex****13.3.5.0.0.33.7 uint8_t enet_config_t::rxAccelerConfig****13.3.5.0.0.33.8 uint8_t enet_config_t::txAccelerConfig****13.3.5.0.0.33.9 uint16_t enet_config_t::pauseDuration****13.3.5.0.0.33.10 uint8_t enet_config_t::rx_fifoEmptyThreshold****13.3.5.0.0.33.11 uint8_t enet_config_t::rx_fifoStatEmptyThreshold**

If the limit is reached, reception continues and a pause frame is triggered.

13.3.5.0.0.33.12 uint8_t enet_config_t::rx_fifoFullThreshold**13.3.5.0.0.33.13 uint8_t enet_config_t::tx_fifoWatermark****13.3.6 struct _enet_handle****Data Fields**

- volatile **enet_rx_bd_struct_t * rxBdBase**
Receive buffer descriptor base address pointer.
- volatile **enet_rx_bd_struct_t * rxBdCurrent**
The current available receive buffer descriptor pointer.
- volatile **enet_tx_bd_struct_t * txBdBase**
Transmit buffer descriptor base address pointer.
- volatile **enet_tx_bd_struct_t * txBdCurrent**
The current available transmit buffer descriptor pointer.
- **uint32_t rxBuffSizeAlign**
Receive buffer size alignment.
- **uint32_t txBuffSizeAlign**
Transmit buffer size alignment.
- **enet_callback_t callback**
Callback function.
- **void * userData**
Callback function parameter.

Macro Definition Documentation

13.3.6.0.0.34 Field Documentation

13.3.6.0.0.34.1 `volatile enet_rx_bd_struct_t* enet_handle_t::rxBdBase`

13.3.6.0.0.34.2 `volatile enet_rx_bd_struct_t* enet_handle_t::rxBdCurrent`

13.3.6.0.0.34.3 `volatile enet_tx_bd_struct_t* enet_handle_t::txBdBase`

13.3.6.0.0.34.4 `volatile enet_tx_bd_struct_t* enet_handle_t::txBdCurrent`

13.3.6.0.0.34.5 `uint32_t enet_handle_t::rxBuffSizeAlign`

13.3.6.0.0.34.6 `uint32_t enet_handle_t::txBuffSizeAlign`

13.3.6.0.0.34.7 `enet_callback_t enet_handle_t::callback`

13.3.6.0.0.34.8 `void* enet_handle_t::userData`

13.4 Macro Definition Documentation

13.4.1 `#define FSL_ENET_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

Version 2.0.1.

Macro Definition Documentation

- 13.4.2 `#define ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK 0x8000U`
- 13.4.3 `#define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK 0x4000U`
- 13.4.4 `#define ENET_BUFFDESCRIPTOR_RX_WRAP_MASK 0x2000U`
- 13.4.5 `#define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask 0x1000U`
- 13.4.6 `#define ENET_BUFFDESCRIPTOR_RX_LAST_MASK 0x0800U`
- 13.4.7 `#define ENET_BUFFDESCRIPTOR_RX_MISS_MASK 0x0100U`
- 13.4.8 `#define ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK 0x0080U`
- 13.4.9 `#define ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK 0x0040U`
- 13.4.10 `#define ENET_BUFFDESCRIPTOR_RX_LENVIOLATE_MASK 0x0020U`
- 13.4.11 `#define ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK 0x0010U`
- 13.4.12 `#define ENET_BUFFDESCRIPTOR_RX_CRC_MASK 0x0004U`
- 13.4.13 `#define ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK 0x0002U`
- 13.4.14 `#define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U`
- 13.4.15 `#define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U`
- 13.4.16 `#define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U`
- 13.4.17 `#define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U`
- 13.4.18 `#define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U`
- 13.4.19 `#define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U`
- 13.4.20 `#define ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK 0x0400U`
- 13.4.21 `#define ENET_BUFFDESCRIPTOR_RX_ERR_MASK`

```
(ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK |
 ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK | \
 ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK |
 ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK |
 ENET_BUFFDESCRIPTOR_RX_CRC_MASK)
```

13.4.22 #define ENET_FRAME_MAX_FRAMELEN 1518U

13.4.23 #define ENET_FRAME_MAX_VALNFRAMELEN 1522U

13.4.24 #define ENET_FIFO_MIN_RX_FULL 5U

13.4.25 #define ENET_RX_MIN_BUFFERSIZE 256U

13.4.26 #define ENET_BUFF_ALIGNMENT 16U

**13.4.27 #define ENET_PHY_MAXADDRESS (ENET_MMFR_PA_MASK >>
ENET_MMFR_PA_SHIFT)**

13.5 Typedef Documentation

**13.5.1 typedef void(* enet_callback_t)(ENET_Type *base, enet_handle_t *handle,
enet_event_t event, void *userData)**

13.6 Enumeration Type Documentation

13.6.1 enum _enet_status

Enumerator

kStatus_ENET_RxFrameError A frame received but data error happen.

kStatus_ENET_RxFrameFail Failed to receive a frame.

kStatus_ENET_RxFrameEmpty No frame arrive.

kStatus_ENET_TxFrameBusy Transmit buffer descriptors are under process.

kStatus_ENET_TxFrameFail Transmit frame fail.

13.6.2 enum enet_mii_mode_t

Enumerator

kENET_MiiMode MII mode for data interface.

kENET_RmiiMode RMII mode for data interface.

Enumeration Type Documentation

13.6.3 enum enet_mii_speed_t

Enumerator

- kENET_MiiSpeed10M* Speed 10 Mbps.
- kENET_MiiSpeed100M* Speed 100 Mbps.

13.6.4 enum enet_mii_duplex_t

Enumerator

- kENET_MiiHalfDuplex* Half duplex mode.
- kENET_MiiFullDuplex* Full duplex mode.

13.6.5 enum enet_mii_write_t

Enumerator

- kENET_MiiWriteNoCompliant* Write frame operation, but not MII-compliant.
- kENET_MiiWriteValidFrame* Write frame operation for a valid MII management frame.

13.6.6 enum enet_mii_read_t

Enumerator

- kENET_MiiReadValidFrame* Read frame operation for a valid MII management frame.
- kENET_MiiReadNoCompliant* Read frame operation, but not MII-compliant.

13.6.7 enum enet_special_control_flag_t

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to macSpecialConfig in the [enet_config_t](#). The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifoFullThreshold and txFifoWatermark in the [enet_config_t](#).

Enumerator

- kENET_ControlFlowControlEnable* Enable ENET flow control: pause frame.
- kENET_ControlRxPayloadCheckEnable* Enable ENET receive payload length check.
- kENET_ControlRxPadRemoveEnable* Padding is removed from received frames.

kENET_ControlRxBroadCastRejectEnable Enable broadcast frame reject.
kENET_ControlMacAddrInsert Enable MAC address insert.
kENET_ControlStoreAndFwdDisable Enable FIFO store and forward.
kENET_ControlSMIPreambleDisable Enable SMI preamble.
kENET_ControlPromiscuousEnable Enable promiscuous mode.
kENET_ControlMIILoopEnable Enable ENET MII loop back.
kENET_ControlVLANTagEnable Enable VLAN tag frame.

13.6.8 enum enet_interrupt_enable_t

This enumeration uses one-bit encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

Enumerator

kENET_BabrInterrupt Babbling receive error interrupt source.
kENET_BabtInterrupt Babbling transmit error interrupt source.
kENET_GraceStopInterrupt Graceful stop complete interrupt source.
kENET_TxFrameInterrupt TX FRAME interrupt source.
kENET_TxBufferInterrupt TX BUFFER interrupt source.
kENET_RxFrameInterrupt RX FRAME interrupt source.
kENET_RxBufferInterrupt RX BUFFER interrupt source.
kENET_MiiInterrupt MII interrupt source.
kENET_EBusERInterrupt Ethernet bus error interrupt source.
kENET_LateCollisionInterrupt Late collision interrupt source.
kENET_RetryLimitInterrupt Collision Retry Limit interrupt source.
kENET_UnderrunInterrupt Transmit FIFO underrun interrupt source.
kENET_PayloadRxInterrupt Payload Receive interrupt source.
kENET_WakeupInterrupt WAKEUP interrupt source.

13.6.9 enum enet_event_t

Enumerator

kENET_RxEvent Receive event.
kENET_TxEvent Transmit event.
kENET_ErrEvent Error event: BABR/BABT/EBERR/LC/RL/UN/PLR .
kENET_WakeUpEvent Wake up from sleep mode event.

Function Documentation

13.6.10 enum enet_tx_accelerator_t

Enumerator

kENET_TxAccelIsShift16Enabled Transmit FIFO shift-16.
kENET_TxAccelIpCheckEnabled Insert IP header checksum.
kENET_TxAccelProtoCheckEnabled Insert protocol checksum.

13.6.11 enum enet_rx_accelerator_t

Enumerator

kENET_RxAccelPadRemoveEnabled Padding removal for short IP frames.
kENET_RxAccelIpCheckEnabled Discard with wrong IP header checksum.
kENET_RxAccelProtoCheckEnabled Discard with wrong protocol checksum.
kENET_RxAccelMacCheckEnabled Discard with Mac layer errors.
kENET_RxAccelIsShift16Enabled Receive FIFO shift-16.

13.7 Function Documentation

13.7.1 void ENET_GetDefaultConfig (enet_config_t * config)

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

```
enet_config_t config;  
ENET_GetDefaultConfig(&config);
```

Parameters

<i>config</i>	The ENET mac controller configuration structure pointer.
---------------	--

13.7.2 void ENET_Init (ENET_Type * base, enet_handle_t * handle, const enet_config_t * config, const enet_buffer_config_t * bufferConfig, uint8_t * macAddr, uint32_t srcClock_Hz)

This function ungates the module clock and initializes it with the ENET configuration.

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	ENET handler pointer.
<i>config</i>	ENET Mac configuration structure pointer. The "enet_config_t" type mac configuration return from ENET_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.
<i>bufferConfig</i>	ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization.
<i>macAddr</i>	ENET mac address of the Ethernet device. This Mac address should be provided.
<i>srcClock_Hz</i>	The internal module clock source for MII clock.

Note

ENET has two buffer descriptors legacy buffer descriptors and enhanced IEEE 1588 buffer descriptors. The legacy descriptor is used by default. To use the IEEE 1588 feature, use the enhanced IEEE 1588 buffer descriptor by defining "ENET_ENHANCEDBUFFERDESCRIPTOR_MODE" and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling [ENET_Init\(\)](#).

13.7.3 void ENET_Deinit (ENET_Type * *base*)

This function gates the module clock, clears ENET interrupts, and disables the ENET module.

Parameters

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

13.7.4 static void ENET_Reset (ENET_Type * *base*) [inline], [static]

This function restores the ENET module to the reset state. Note that this function sets all registers to the reset state. As a result, the ENET module can't work after calling this function.

Parameters

Function Documentation

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

13.7.5 void ENET_SetMII (ENET_Type * *base*, enet_mii_speed_t *speed*, enet_mii_duplex_t *duplex*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>speed</i>	The speed of the RMII mode.
<i>duplex</i>	The duplex of the RMII mode.

13.7.6 void ENET_SetSMI (ENET_Type * *base*, uint32_t *srcClock_Hz*, bool *isPreambleDisabled*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>srcClock_Hz</i>	This is the ENET module clock frequency. Normally it's the system clock. See clock distribution.
<i>isPreambleDisabled</i>	The preamble disable flag. <ul style="list-style-type: none">• true Enables the preamble.• false Disables the preamble.

13.7.7 static bool ENET_GetSMI (ENET_Type * *base*) [inline], [static]

This API is used to get the SMI configuration to check whether the MII management interface has been set.

Parameters

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

Returns

The SMI setup status true or false.

13.7.8 **static uint32_t ENET_ReadSMIData (ENET_Type * *base*) [inline],
[static]**

Function Documentation

Parameters

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

Returns

The data read from PHY

13.7.9 void ENET_StartSMIRead (ENET_Type * *base*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_read_t *operation*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>phyAddr</i>	The PHY address.
<i>phyReg</i>	The PHY register.
<i>operation</i>	The read operation.

13.7.10 void ENET_StartSMIWrite (ENET_Type * *base*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_write_t *operation*, uint32_t *data*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>phyAddr</i>	The PHY address.
<i>phyReg</i>	The PHY register.
<i>operation</i>	The write operation.
<i>data</i>	The data written to PHY.

13.7.11 void ENET_SetMacAddr (ENET_Type * *base*, uint8_t * *macAddr*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>macAddr</i>	The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

13.7.12 void ENET_GetMacAddr (ENET_Type * *base*, uint8_t * *macAddr*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>macAddr</i>	The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

13.7.13 void ENET_AddMulticastGroup (ENET_Type * *base*, uint8_t * *address*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>address</i>	The six-byte multicast group address which is provided by application.

13.7.14 void ENET_LeaveMulticastGroup (ENET_Type * *base*, uint8_t * *address*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>address</i>	The six-byte multicast group address which is provided by application.

13.7.15 static void ENET_ActiveRead (ENET_Type * *base*) [inline], [static]

Function Documentation

Parameters

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

Note

This must be called after the MAC configuration and state are ready. It must be called after the [ENET_Init\(\)](#) and [ENET_Ptp1588Configure\(\)](#). This should be called when the ENET receive required.

13.7.16 static void ENET_EnableSleepMode (ENET_Type * *base*, bool *enable*) [inline], [static]

This function is used to set the MAC enter sleep mode. When entering sleep mode, the magic frame wakeup interrupt should be enabled to wake up MAC from the sleep mode and reset it to normal mode.

Parameters

<i>base</i>	ENET peripheral base address.
<i>enable</i>	True enable sleep mode, false disable sleep mode.

13.7.17 static void ENET_GetAccelFunction (ENET_Type * *base*, uint32_t * *txAccelOption*, uint32_t * *rxAccelOption*) [inline], [static]

Parameters

<i>base</i>	ENET peripheral base address.
<i>txAccelOption</i>	The transmit accelerator option. The "enet_tx_accelerator_t" is recommended as the mask to get the exact the accelerator option.
<i>rxAccelOption</i>	The receive accelerator option. The "enet_rx_accelerator_t" is recommended as the mask to get the exact the accelerator option.

13.7.18 static void ENET_EnableInterrupts (ENET_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enumeration members. See [enet_interrupt_enable_t](#). For example, to enable the TX frame interrupt and RX frame interrupt, do the following.

```
*     ENET_EnableInterrupts(ENET, kENET_TxFrameInterrupt |  
*                           kENET_RxFrameInterrupt);  
*
```

Function Documentation

Parameters

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

13.7.19 static void ENET_DisableInterrupts (ENET_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [enet_interrupt_enable_t](#). For example, to disable the TX frame interrupt and RX frame interrupt, do the following.

```
*     ENET_DisableInterrupts(ENET, kENET_TxFrameInterrupt |  
*                           kENET_RxFrameInterrupt);  
*
```

Parameters

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

13.7.20 static uint32_t ENET_GetInterruptStatus (ENET_Type * *base*) [inline], [static]

Parameters

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

Returns

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

13.7.21 static void ENET_ClearInterruptStatus (ENET_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears enabled ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See the [enet_interrupt_enable_t](#). For example, to clear the TX frame interrupt and RX frame interrupt, do the following.

```
*     ENET_ClearInterruptStatus(ENET,
*                                kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
*
```

Parameters

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

13.7.22 void ENET_SetCallback (*enet_handle_t* * *handle*, *enet_callback_t* *callback*, *void* * *userData*)

This API is provided for the application callback required case when ENET interrupt is enabled. This API should be called after calling ENET_Init.

Parameters

<i>handle</i>	ENET handler pointer. Should be provided by application.
<i>callback</i>	The ENET callback function.
<i>userData</i>	The callback function parameter.

13.7.23 void ENET_GetRxErrBeforeReadFrame (*enet_handle_t* * *handle*, *enet_data_error_stats_t* * *eErrorStatic*)

This API must be called after the ENET_GetRxFrameSize and before the [ENET_ReadFrame\(\)](#). If the ENET_GetRxFrameSize returns kStatus_ENET_RxFrameError, the ENET_GetRxErrBeforeReadFrame can be used to get the exact error statistics. This is an example.

```
*     status = ENET_GetRxFrameSize(&g_handle, &length);
*     if (status == kStatus_ENET_RxFrameError)
*     {
*         // Get the error information of the received frame.
*         ENET_GetRxErrBeforeReadFrame(&g_handle, &eErrStatic);
*         // update the receive buffer.
*         ENET_ReadFrame(EXAMPLE_ENET, &g_handle, NULL, 0);
*     }
*
```

Function Documentation

Parameters

<i>handle</i>	The ENET handler structure pointer. This is the same handler pointer used in the ENET_Init.
<i>eErrorStatic</i>	The error statistics structure pointer.

13.7.24 status_t ENET_GetRxFrameSize (enet_handle_t * *handle*, uint32_t * *length*)

This function gets a received frame size from the ENET buffer descriptors.

Note

The FCS of the frame is automatically removed by Mac and the size is the length without the FCS. After calling ENET_GetRxFrameSize, [ENET_ReadFrame\(\)](#) should be called to update the receive buffers If the result is not "kStatus_ENET_RxFrameEmpty".

Parameters

<i>handle</i>	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
<i>length</i>	The length of the valid frame received.

Return values

<i>kStatus_ENET_RxFrame-Empty</i>	No frame received. Should not call ENET_ReadFrame to read frame.
<i>kStatus_ENET_RxFrame-Error</i>	Data error happens. ENET_ReadFrame should be called with NULL data and NULL length to update the receive buffers.
<i>kStatus_Success</i>	Receive a frame Successfully then the ENET_ReadFrame should be called with the right data buffer and the captured data length input.

13.7.25 status_t ENET_ReadFrame (ENET_Type * *base*, enet_handle_t * *handle*, uint8_t * *data*, uint32_t *length*)

This function reads a frame (both the data and the length) from the ENET buffer descriptors. The ENET_GetRxFrameSize should be used to get the size of the prepared data buffer. This is an example.

```
*     uint32_t length;
*     enet_handle_t g_handle;
*     //Get the received frame size firstly.
*     status = ENET_GetRxFrameSize(&g_handle, &length);
```

```

*     if (length != 0)
*     {
*         //Allocate memory here with the size of "length"
*         uint8_t *data = memory allocate interface;
*         if (!data)
*         {
*             ENET_ReadFrame(ENET, &g_handle, NULL, 0);
*             //Add the console warning log.
*         }
*         else
*         {
*             status = ENET_ReadFrame(ENET, &g_handle, data, length);
*             //Call stack input API to deliver the data to stack
*         }
*     }
*     else if (status == kStatus_ENET_RxFrameError)
*     {
*         //Update the received buffer when a error frame is received.
*         ENET_ReadFrame(ENET, &g_handle, NULL, 0);
*     }
*

```

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET handler structure. This is the same handler pointer used in the ENET_Init.
<i>data</i>	The data buffer provided by user to store the frame which memory size should be at least "length".
<i>length</i>	The size of the data buffer which is still the length of the received frame.

Returns

The execute status, successful or failure.

13.7.26 status_t ENET_SendFrame (ENET_Type * *base*, enet_handle_t * *handle*, uint8_t * *data*, uint32_t *length*)

Note

The CRC is automatically appended to the data. Input the data to send without the CRC.

Parameters

Function Documentation

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
<i>data</i>	The data buffer provided by user to be send.
<i>length</i>	The length of the data to be send.

Return values

<i>kStatus_Success</i>	Send frame succeed.
<i>kStatus_ENET_TxFrameBusy</i>	Transmit buffer descriptor is busy under transmission. The transmit busy happens when the data send rate is over the MAC capacity. The waiting mechanism is recommended to be added after each call return with <i>kStatus_ENET_TxFrameBusy</i> .

13.7.27 void ENET_TransmitIRQHandler (ENET_Type * *base*, enet_handle_t * *handle*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET handler pointer.

13.7.28 void ENET_ReceiveIRQHandler (ENET_Type * *base*, enet_handle_t * *handle*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET handler pointer.

13.7.29 void ENET_ErrorIRQHandler (ENET_Type * *base*, enet_handle_t * *handle*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET handler pointer.

Function Documentation

Chapter 14

EWM: External Watchdog Monitor Driver

14.1 Overview

The KSDK provides a peripheral driver for the EWM module of Kinetis devices.

14.2 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

- static uint32_t [EWM_GetStatusFlags](#) (EWM_Type *base)
Disables the EWM interrupt.
- void [EWM_Refresh](#) (EWM_Type *base)
Gets all status flags.
- Services the EWM.

14.3 Data Structure Documentation

14.3.1 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.

14.4 Macro Definition Documentation

14.4.1 #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

14.5 Enumeration Type Documentation

14.5.1 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.

14.5.2 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.

14.6 Function Documentation

14.6.1 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

14.6.2 void EWM_Deinit (EWM_Type * *base*)

This function is used to shut down the EWM.

Parameters

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

14.6.3 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](#)

14.6.4 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

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

This function disables 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

14.6.6 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.

14.6.7 void EWM_Refresh (EWM_Type * *base*)

This function resets the EWM counter to zero.

Parameters

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

Function Documentation

Chapter 15

C90TFS Flash Driver

15.1 Overview

The flash provides the C90TFS Flash driver of Kinetis devices with the C90TFS Flash module inside. The flash driver provides general APIs to handle specific operations on C90TFS/FTFx Flash module. The user can use those APIs directly in the application. In addition, it provides internal functions called by the driver. Although these functions are not meant to be called from the user's application directly, the APIs can still be used.

Data Structures

- struct [flash_execute_in_ram_function_config_t](#)
Flash execute-in-RAM function information. [More...](#)
- struct [flash_swap_state_config_t](#)
Flash Swap information. [More...](#)
- struct [flash_swap_ifr_field_config_t](#)
Flash Swap IFR fields. [More...](#)
- union [flash_swap_ifr_field_data_t](#)
Flash Swap IFR field data. [More...](#)
- union [pflash_protection_status_low_t](#)
PFlash protection status - low 32bit. [More...](#)
- struct [pflash_protection_status_t](#)
PFlash protection status - full. [More...](#)
- struct [flash_prefetch_speculation_status_t](#)
Flash prefetch speculation status. [More...](#)
- struct [flash_protection_config_t](#)
Active flash protection information for the current operation. [More...](#)
- struct [flash_access_config_t](#)
Active flash Execute-Only access information for the current operation. [More...](#)
- struct [flash_operation_config_t](#)
Active flash information for the current operation. [More...](#)
- struct [flash_config_t](#)
Flash driver state information. [More...](#)

Typedefs

- [typedef void\(* flash_callback_t \)\(void\)](#)
A callback type used for the Pflash block.

Enumerations

- enum [flash_margin_value_t](#) {
 kFLASH_MarginValueNormal,
 kFLASH_MarginValueUser,
 kFLASH_MarginValueFactory,

Overview

`kFLASH_MarginValueInvalid }`

Enumeration for supported flash margin levels.

- enum `flash_security_state_t` {
 `kFLASH_SecurityStateNotSecure,`
 `kFLASH_SecurityStateBackdoorEnabled,`
 `kFLASH_SecurityStateBackdoorDisabled }`

Enumeration for the three possible flash security states.

- enum `flash_protection_state_t` {
 `kFLASH_ProtectionStateUnprotected,`
 `kFLASH_ProtectionStateProtected,`
 `kFLASH_ProtectionStateMixed }`

Enumeration for the three possible flash protection levels.

- enum `flash_execute_only_access_state_t` {
 `kFLASH_AccessStateUnLimited,`
 `kFLASH_AccessStateExecuteOnly,`
 `kFLASH_AccessStateMixed }`

Enumeration for the three possible flash execute access levels.

- enum `flash_property_tag_t` {
 `kFLASH_PropertyPflashSectorSize = 0x00U,`
 `kFLASH_PropertyPflashTotalSize = 0x01U,`
 `kFLASH_PropertyPflashBlockSize = 0x02U,`
 `kFLASH_PropertyPflashBlockCount = 0x03U,`
 `kFLASH_PropertyPflashBlockBaseAddr = 0x04U,`
 `kFLASH_PropertyPflashFacSupport = 0x05U,`
 `kFLASH_PropertyPflashAccessSegmentSize = 0x06U,`
 `kFLASH_PropertyPflashAccessSegmentCount = 0x07U,`
 `kFLASH_PropertyFlexRamBlockBaseAddr = 0x08U,`
 `kFLASH_PropertyFlexRamTotalSize = 0x09U,`
 `kFLASH_PropertyDflashSectorSize = 0x10U,`
 `kFLASH_PropertyDflashTotalSize = 0x11U,`
 `kFLASH_PropertyDflashBlockSize = 0x12U,`
 `kFLASH_PropertyDflashBlockCount = 0x13U,`
 `kFLASH_PropertyDflashBlockBaseAddr = 0x14U,`
 `kFLASH_PropertyEepromTotalSize = 0x15U,`
 `kFLASH_PropertyFlashMemoryIndex = 0x20U }`

Enumeration for various flash properties.

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

Constants for execute-in-RAM flash function.

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

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

- enum `_flash_read_resource_range` {

```

kFLASH_ResourceRangePflashIfrSizeInBytes = 1024U,
kFLASH_ResourceRangeVersionIdSizeInBytes = 8U,
kFLASH_ResourceRangeVersionIdStart = 0x08U,
kFLASH_ResourceRangeVersionIdEnd = 0x0FU,
kFLASH_ResourceRangePflashSwapIfrStart = 0x40000U,
kFLASH_ResourceRangePflashSwapIfrEnd,
kFLASH_ResourceRangeDflashIfrStart = 0x800000U,
kFLASH_ResourceRangeDflashIfrEnd = 0x8003FFU }

```

Enumeration for the range of special-purpose flash resource.

- enum `flash_flexram_function_option_t` {
 kFLASH_FlexramFunctionOptionAvailableAsRam = 0xFFU,
 kFLASH_FlexramFunctionOptionAvailableForEeprom = 0x00U }

Enumeration for the two possible options of set FlexRAM function command.

- enum `_flash_acceleration_ram_property`

Enumeration for acceleration RAM property.

- enum `flash_swap_function_option_t` {
 kFLASH_SwapFunctionOptionEnable = 0x00U,
 kFLASH_SwapFunctionOptionDisable = 0x01U }

Enumeration for the possible options of Swap function.

- enum `flash_swap_control_option_t` {
 kFLASH_SwapControlOptionInitializeSystem = 0x01U,
 kFLASH_SwapControlOptionSetInUpdateState = 0x02U,
 kFLASH_SwapControlOptionSetInCompleteState = 0x04U,
 kFLASH_SwapControlOptionReportStatus = 0x08U,
 kFLASH_SwapControlOptionDisableSystem = 0x10U }

Enumeration for the possible options of Swap control commands.

- enum `flash_swap_state_t` {
 kFLASH_SwapStateUninitialized = 0x00U,
 kFLASH_SwapStateReady = 0x01U,
 kFLASH_SwapStateUpdate = 0x02U,
 kFLASH_SwapStateUpdateErased = 0x03U,
 kFLASH_SwapStateComplete = 0x04U,
 kFLASH_SwapStateDisabled = 0x05U }

Enumeration for the possible flash Swap status.

- enum `flash_swap_block_status_t` {
 kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero,
 kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero }

Enumeration for the possible flash Swap block status

- enum `flash_partition_flexram_load_option_t` {
 kFLASH_PartitionFlexramLoadOptionLoadedWithValidEepromData,
 kFLASH_PartitionFlexramLoadOptionNotLoaded = 0x01U }

Enumeration for the FlexRAM load during reset option.

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

Enumeration for the flash memory index.

- enum `flash_prefetch_speculation_option_t`

Overview

Enumeration for the two possible options of flash prefetch speculation.

Flash version

- enum `_flash_driver_version_constants` {
 `kFLASH_DriverVersionName` = 'F',
 `kFLASH_DriverVersionMajor` = 2,
 `kFLASH_DriverVersionMinor` = 2,
 `kFLASH_DriverVersionBugfix` = 0 }
 Flash driver version for ROM.
- #define `MAKE_VERSION`(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))
 Constructs the version number for drivers.
- #define `FSL_FLASH_DRIVER_VERSION` (`MAKE_VERSION`(2, 2, 0))
 Flash driver version for SDK.

Flash configuration

- #define `FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT` 1
 Indicates whether to support FlexNVM in the Flash driver.
- #define `FLASH_SSD_IS_FLEXNVM_ENABLED` (`FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT` && `FSL_FEATURE_FLASH_HAS_FLEX_NVM`)
 Indicates whether the FlexNVM is enabled in the Flash driver.
- #define `FLASH_SSD_IS_SECONDARY_FLASH_SUPPORTED` (0)
 Indicates whether the secondary flash is supported in the Flash driver.
- #define `FLASH_SSD_SECONDARY_FLASH_HAS_ITS_OWN_PROTECTION_REGISTER` (0)
 Indicates whether the secondary flash has its own protection register in flash module.
- #define `FLASH_SSD_SECONDARY_FLASH_HAS_ITS_OWN_ACCESS_REGISTER` (0)
 Indicates whether the secondary flash has its own Execute-Only access register in flash module.
- #define `FLASH_DRIVER_IS_FLASH_RESIDENT` 1
 Flash driver location.
- #define `FLASH_DRIVER_IS_EXPORTED` 0
 Flash Driver Export option.

Flash status

- enum `_flash_status` {

`kStatus_FLASH_Success` = MAKE_STATUS(kStatusGroupGeneric, 0),

`kStatus_FLASH_InvalidArgument` = MAKE_STATUS(kStatusGroupGeneric, 4),

`kStatus_FLASH_SizeError` = MAKE_STATUS(kStatusGroupFlashDriver, 0),

`kStatus_FLASH_AlignmentError`,

`kStatus_FLASH_AddressError` = MAKE_STATUS(kStatusGroupFlashDriver, 2),

`kStatus_FLASH_AccessError`,

`kStatus_FLASH_ProtectionViolation`,

`kStatus_FLASH_CommandFailure`,

`kStatus_FLASH_UnknownProperty` = MAKE_STATUS(kStatusGroupFlashDriver, 6),

`kStatus_FLASH_EraseKeyError` = MAKE_STATUS(kStatusGroupFlashDriver, 7),

`kStatus_FLASH_RegionExecuteOnly`,

`kStatus_FLASH_ExecuteInRamFunctionNotReady`,

`kStatus_FLASH_PartitionStatusUpdateFailure`,

`kStatus_FLASH_SetFlexramAsEepromError`,

`kStatus_FLASH_RecoverFlexramAsRamError`,

`kStatus_FLASH_SetFlexramAsRamError` = MAKE_STATUS(kStatusGroupFlashDriver, 13),

`kStatus_FLASH_RecoverFlexramAsEepromError`,

`kStatus_FLASH_CommandNotSupported` = MAKE_STATUS(kStatusGroupFlashDriver, 15),

`kStatus_FLASH_SwapSystemNotInUninitialized`,

`kStatus_FLASH_SwapIndicatorAddressError`,

`kStatus_FLASH_ReadOnlyProperty` = MAKE_STATUS(kStatusGroupFlashDriver, 18),

`kStatus_FLASH_InvalidPropertyValue`,

`kStatus_FLASH_InvalidSpeculationOption` }

Flash driver status codes.
- #define `kStatusGroupGeneric` 0

Flash driver status group.
- #define `kStatusGroupFlashDriver` 1

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

Constructs a status code value from a group and a code number.

Flash API key

- enum `_flash_driver_api_keys` { `kFLASH_ApiEraseKey` = FOUR_CHAR_CODE('k', 'f', 'e', 'k') }

Enumeration for Flash driver API keys.
- #define `FOUR_CHAR_CODE`(a, b, c, d) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))

Constructs the four character code for the Flash driver API key.

Initialization

- `status_t FLASH_Init (flash_config_t *config)`

Initializes the global flash properties structure members.
- `status_t FLASH_SetCallback (flash_config_t *config, flash_callback_t callback)`

Sets the desired flash callback function.
- `status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t *config)`

Prepares flash execute-in-RAM functions.

Overview

Erasing

- status_t **FLASH_EraseAll** (**flash_config_t** *config, uint32_t key)
Erases entire flash.
- status_t **FLASH_Erase** (**flash_config_t** *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)
Erases the flash sectors encompassed by parameters passed into function.
- status_t **FLASH_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.
- status_t **FLASH_ProgramSection** (**flash_config_t** *config, uint32_t start, uint32_t *src, uint32_t lengthInBytes)
Programs flash with data at locations passed in through parameters via the Program Section command.
- status_t **FLASH_EepromWrite** (**flash_config_t** *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)
Programs the EEPROM with data at locations passed in through parameters.

Reading

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

Security

- status_t **FLASH_GetSecurityState** (**flash_config_t** *config, **flash_security_state_t** *state)
Returns the security state via the pointer passed into the function.
- status_t **FLASH_SecurityBypass** (**flash_config_t** *config, const uint8_t *backdoorKey)
Allows users to bypass security with a backdoor key.

Verification

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

- Verifies programming of the desired flash area at a specified margin level.
- status_t `FLASH_VerifyEraseAllExecuteOnlySegments` (`flash_config_t` *config, `flash_margin_value_t` margin)

Verifies whether the program flash execute-only segments have been erased to the specified read margin level.

Protection

- status_t `FLASH_IsProtected` (`flash_config_t` *config, `uint32_t` start, `uint32_t` lengthInBytes, `flash_protection_state_t` *protection_state)

Returns the protection state of the desired flash area via the pointer passed into the function.
- status_t `FLASH_IsExecuteOnly` (`flash_config_t` *config, `uint32_t` start, `uint32_t` lengthInBytes, `flash_execute_only_access_state_t` *access_state)

Returns the access state of the desired flash area via the pointer passed into the function.

Properties

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

Returns the desired flash property.

FlexRAM

Sets the desired flash property.

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 <code>flash_property_tag_t</code>
<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.

Overview

<code>kStatus_FLASH_Invalid- PropertyValue</code>	An invalid property value.
<code>kStatus_FLASH_Read- OnlyProperty</code>	An read-only property tag.

- `status_t FLASH_SetFlexramFunction (flash_config_t *config, flash flexram_function_option_t option)`
Sets the FlexRAM function command.

Swap

- `status_t FLASH_SwapControl (flash_config_t *config, uint32_t address, flash_swap_control_option_t option, flash_swap_state_config_t *returnInfo)`
Configures the Swap function or checks the swap state of the Flash module.
- `status_t FLASH_Swap (flash_config_t *config, uint32_t address, flash_swap_function_option_t option)`
Swaps the lower half flash with the higher half flash.

FlexNVM

- `status_t FLASH_ProgramPartition (flash_config_t *config, flash_partition_flexram_load_option_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)`
Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Flash Protection Utilities

- `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.
- `status_t FLASH_DflashSetProtection (flash_config_t *config, uint8_t protectStatus)`
Sets the DFlash protection to the intended protection status.
- `status_t FLASH_DflashGetProtection (flash_config_t *config, uint8_t *protectStatus)`
Gets the DFlash protection status.
- `status_t FLASH_EepromSetProtection (flash_config_t *config, uint8_t protectStatus)`
Sets the EEPROM protection to the intended protection status.
- `status_t FLASH_EepromGetProtection (flash_config_t *config, uint8_t *protectStatus)`
Gets the DFlash protection status.

15.2 Data Structure Documentation

15.2.1 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.

15.2.1.0.0.35 Field Documentation

15.2.1.0.0.35.1 `uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount`

15.2.1.0.0.35.2 `uint32_t* flash_execute_in_ram_function_config_t::flashRunCommand`

15.2.1.0.0.35.3 `uint32_t* flash_execute_in_ram_function_config_t::flashCommonBitOperation`

15.2.2 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.

15.2.2.0.0.36 Field Documentation

15.2.2.0.0.36.1 `flash_swap_state_t flash_swap_state_config_t::flashSwapState`

15.2.2.0.0.36.2 `flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus`

15.2.2.0.0.36.3 `flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus`

15.2.3 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]`

Data Structure Documentation

- `uint8_t reserved1 [2]`
A reserved field.
- `uint16_t swapDisableWord`
A Swap disable word field.
- `uint8_t reserved2 [4]`
A reserved field.

15.2.3.0.0.37 Field Documentation

15.2.3.0.0.37.1 `uint16_t flash_swap_ifr_field_config_t::swapIndicatorAddress`

15.2.3.0.0.37.2 `uint16_t flash_swap_ifr_field_config_t::swapEnableWord`

15.2.3.0.0.37.3 `uint8_t flash_swap_ifr_field_config_t::reserved0[4]`

15.2.3.0.0.37.4 `uint8_t flash_swap_ifr_field_config_t::reserved1[2]`

15.2.3.0.0.37.5 `uint16_t flash_swap_ifr_field_config_t::swapDisableWord`

15.2.3.0.0.37.6 `uint8_t flash_swap_ifr_field_config_t::reserved2[4]`

15.2.4 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.

15.2.4.0.0.38 Field Documentation

15.2.4.0.0.38.1 `uint32_t flash_swap_ifr_field_data_t::flashSwapIfrData[2]`

15.2.4.0.0.38.2 `flash_swap_ifr_field_config_t flash_swap_ifr_field_data_t::flashSwapIfrField`

15.2.5 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] .

15.2.5.0.0.39 Field Documentation

15.2.5.0.0.39.1 `uint32_t pflash_protection_status_low_t::protl32b`

15.2.5.0.0.39.2 `uint8_t pflash_protection_status_low_t::protsl`

15.2.5.0.0.39.3 `uint8_t pflash_protection_status_low_t::protsh`

15.2.6 struct pflash_protection_status_t**Data Fields**

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

15.2.6.0.0.40 Field Documentation

15.2.6.0.0.40.1 `pflash_protection_status_low_t pflash_protection_status_t::valueLow32b`

15.2.7 struct flash_prefetch_speculation_status_t**Data Fields**

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

15.2.7.0.0.41 Field Documentation

15.2.7.0.0.41.1 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::instructionOption`

15.2.7.0.0.41.2 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::dataOption`

15.2.8 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.

Data Structure Documentation

15.2.8.0.0.42 Field Documentation

15.2.8.0.0.42.1 `uint32_t flash_protection_config_t::regionBase`

15.2.8.0.0.42.2 `uint32_t flash_protection_config_t::regionSize`

15.2.8.0.0.42.3 `uint32_t flash_protection_config_t::regionCount`

15.2.9 `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.

15.2.9.0.0.43 Field Documentation

15.2.9.0.0.43.1 `uint32_t flash_access_config_t::SegmentBase`

15.2.9.0.0.43.2 `uint32_t flash_access_config_t::SegmentSize`

15.2.9.0.0.43.3 `uint32_t flash_access_config_t::SegmentCount`

15.2.10 `struct flash_operation_config_t`

Data Fields

- `uint32_t convertedAddress`
A converted address for the current flash type.
- `uint32_t activeSectorSize`
A sector size of the current flash type.
- `uint32_t activeBlockSize`
A block size of the current flash type.
- `uint32_t blockWriteUnitSize`
The write unit size.
- `uint32_t sectorCmdAddressAlignment`
An erase sector command address alignment.
- `uint32_t partCmdAddressAlignment`
A program/verify part command address alignment.
- `32_t resourceCmdAddressAlignment`
A read resource command address alignment.
- `uint32_t checkCmdAddressAlignment`
A program check command address alignment.

15.2.10.0.0.44 Field Documentation

- 15.2.10.0.0.44.1 `uint32_t flash_operation_config_t::convertedAddress`
- 15.2.10.0.0.44.2 `uint32_t flash_operation_config_t::activeSectorSize`
- 15.2.10.0.0.44.3 `uint32_t flash_operation_config_t::activeBlockSize`
- 15.2.10.0.0.44.4 `uint32_t flash_operation_config_t::blockWriteUnitSize`
- 15.2.10.0.0.44.5 `uint32_t flash_operation_config_t::sectorCmdAddressAlignment`
- 15.2.10.0.0.44.6 `uint32_t flash_operation_config_t::partCmdAddressAlignment`
- 15.2.10.0.0.44.7 `uint32_t flash_operation_config_t::resourceCmdAddressAlignment`
- 15.2.10.0.0.44.8 `uint32_t flash_operation_config_t::checkCmdAddressAlignment`

15.2.11 `struct flash_config_t`

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

Data Fields

- `uint32_t PFlashBlockBase`
A base address of the first PFlash block.
- `uint32_t PFlashTotalSize`
The size of the combined PFlash block.
- `uint32_t PFlashBlockCount`
A number of PFlash blocks.
- `uint32_t PFlashSectorSize`
The size in bytes of a sector of PFlash.
- `flash_callback_t PFlashCallback`
The callback function for the flash API.
- `uint32_t PFlashAccessSegmentSize`
A size in bytes of an access segment of PFlash.
- `uint32_t PFlashAccessSegmentCount`
A number of PFlash access segments.
- `uint32_t * flashExecuteInRamFunctionInfo`
An information structure of the flash execute-in-RAM function.
- `uint32_t FlexRAMBlockBase`
For the FlexNVM device, this is the base address of the FlexRAM For the non-FlexNVM device, this is the base address of the acceleration RAM memory.
- `uint32_t FlexRAMTotalSize`
For the FlexNVM device, this is the size of the FlexRAM For the non-FlexNVM device, this is the size of the acceleration RAM memory.
- `uint32_t DFlashBlockBase`
For the FlexNVM device, this is the base address of the D-Flash memory (FlexNVM memory) For the

Macro Definition Documentation

- *non-FlexNVM device, this field is unused.*
- **uint32_t DFlashTotalSize**
For the FlexNVM device, this is the total size of the FlexNVM memory; For the non-FlexNVM device, this field is unused.
- **uint32_t EEepromTotalSize**
For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM For the non-FlexNVM device, this field is unused.
- **uint32_t FlashMemoryIndex**
0 - primary flash; 1 - secondary flash

15.2.11.0.0.45 Field Documentation

15.2.11.0.0.45.1 uint32_t flash_config_t::PFlashTotalSize

15.2.11.0.0.45.2 uint32_t flash_config_t::PFlashBlockCount

15.2.11.0.0.45.3 uint32_t flash_config_t::PFlashSectorSize

15.2.11.0.0.45.4 flash_callback_t flash_config_t::PFlashCallback

15.2.11.0.0.45.5 uint32_t flash_config_t::PFlashAccessSegmentSize

15.2.11.0.0.45.6 uint32_t flash_config_t::PFlashAccessSegmentCount

15.2.11.0.0.45.7 uint32_t* flash_config_t::flashExecuteInRamFunctionInfo

15.3 Macro Definition Documentation

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

15.3.2 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))

Version 2.2.0.

15.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enables the FlexNVM support by default.

15.3.4 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

15.3.5 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the KSDK application.

15.3.6 #define kStatusGroupGeneric 0

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

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

15.4 Enumeration Type Documentation

15.4.1 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.

15.4.2 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.

Enumeration Type Documentation

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.

15.4.3 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.

15.4.4 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.

15.4.5 enum flash_security_state_t

Enumerator

kFLASH_SecurityStateNotSecure Flash is not secure.
kFLASH_SecurityStateBackdoorEnabled Flash backdoor is enabled.
kFLASH_SecurityStateBackdoorDisabled Flash backdoor is disabled.

15.4.6 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.

15.4.7 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.

15.4.8 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.

Enumeration Type Documentation

15.4.9 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.

15.4.10 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.

15.4.11 enum _flash_read_resource_range

Enumerator

kFLASH_ResourceRangePflashIfrSizeInBytes Pflash IFR size in byte.
kFLASH_ResourceRangeVersionIdSizeInBytes Version ID IFR size in byte.
kFLASH_ResourceRangeVersionIdStart Version ID IFR start address.
kFLASH_ResourceRangeVersionIdEnd Version ID IFR end address.
kFLASH_ResourceRangePflashSwapIfrStart Pflash swap IFR start address.
kFLASH_ResourceRangePflashSwapIfrEnd Pflash swap IFR end address.
kFLASH_ResourceRangeDflashIfrStart Dflash IFR start address.
kFLASH_ResourceRangeDflashIfrEnd Dflash IFR end address.

15.4.12 enum flash_flexram_function_option_t

Enumerator

kFLASH_FlexramFunctionOptionAvailableAsRam An option used to make FlexRAM available as RAM.
kFLASH_FlexramFunctionOptionAvailableForEeprom An option used to make FlexRAM available for EEPROM.

15.4.13 enum flash_swap_function_option_t

Enumerator

kFLASH_SwapFunctionOptionEnable An option used to enable the Swap function.*kFLASH_SwapFunctionOptionDisable* An option used to disable the Swap function.**15.4.14 enum flash_swap_control_option_t**

Enumerator

kFLASH_SwapControlOptionInitializeSystem An option used to initialize the Swap system.*kFLASH_SwapControlOptionSetInUpdateState* An option used to set the Swap in an update state.*kFLASH_SwapControlOptionSetIncompleteState* An option used to set the Swap in a complete state.*kFLASH_SwapControlOptionReportStatus* An option used to report the Swap status.*kFLASH_SwapControlOptionDisableSystem* An option used to disable the Swap status.**15.4.15 enum flash_swap_state_t**

Enumerator

kFLASH_SwapStateUninitialized Flash Swap system is in an uninitialized state.*kFLASH_SwapStateReady* Flash Swap system is in a ready state.*kFLASH_SwapStateUpdate* Flash Swap system is in an update state.*kFLASH_SwapStateUpdateErased* Flash Swap system is in an updateErased state.*kFLASH_SwapStateComplete* Flash Swap system is in a complete state.*kFLASH_SwapStateDisabled* Flash Swap system is in a disabled state.**15.4.16 enum flash_swap_block_status_t**

Enumerator

kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero Swap block status is that lower half program block at zero.*kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero* Swap block status is that upper half program block at zero.

Function Documentation

15.4.17 enum flash_partition_flexram_load_option_t

Enumerator

kFLASH_PartitionFlexramLoadOptionLoadedWithValidEepromData FlexRAM is loaded with valid EEPROM data during reset sequence.

kFLASH_PartitionFlexramLoadOptionNotLoaded FlexRAM is not loaded during reset sequence.

15.4.18 enum flash_memory_index_t

Enumerator

kFLASH_MemoryIndexPrimaryFlash Current flash memory is primary flash.

kFLASH_MemoryIndexSecondaryFlash Current flash memory is secondary flash.

15.5 Function Documentation

15.5.1 status_t FLASH_Init (flash_config_t * config)

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

Parameters

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

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

15.5.2 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.

15.5.3 status_t FLASH_PreparesExecuteInRamFunctions (flash_config_t * *config*)

Parameters

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

Return values

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

15.5.4 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.

Function Documentation

<i>kStatus_FLASH_Erase- KeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_Execute- InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access- Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_- ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_- CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_- PartitionStatusUpdate- Failure</i>	Failed to update the partition status.

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

15.5.6 **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_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.

Function Documentation

<i>kStatus_FLASH_Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during 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.

15.5.7 **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.

15.5.8 **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.

Function Documentation

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>index</i>	The index indicating which area of the Program Once Field to be programmed.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the Program Once Field.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

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

15.5.9 **status_t FLASH_ProgramSection (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 length.

Parameters

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

Return values

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

15.5.10 **status_t FLASH_EepromWrite (flash_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)**

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

Parameters

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

Function Documentation

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_SetFlexramAsEepromError</i>	Failed to set flexram as eeprom.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_RecoverFlexramAsRamError</i>	Failed to recover the FlexRAM as RAM.

15.5.11 **status_t FLASH_ReadResource (flash_config_t * config, uint32_t start, uint32_t * dst, uint32_t lengthInBytes, flash_read_resource_option_t option)**

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

Parameters

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

Return values

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

15.5.12 **status_t FLASH_ReadOnce (flash_config_t * config, uint32_t index, uint32_t * dst, uint32_t lengthInBytes)**

This function reads the read once feild with given index and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>index</i>	The index indicating the area of program once field to be read.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Function Documentation

Return values

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

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

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

Parameters

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

Return values

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

15.5.14 **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.

15.5.15 **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.

Function Documentation

<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

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

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

Parameters

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

Return values

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

<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.

15.5.17 status_t FLASH_VerifyProgram (*flash_config_t * config*, *uint32_t start*, *uint32_t lengthInBytes*, *const uint32_t * expectedData*, *flash_margin_value_t margin*, *uint32_t * failedAddress*, *uint32_t * failedData*)

This function verifies the data programmed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be verified. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<i>expectedData</i>	A pointer to the expected data that is to be verified against.
<i>margin</i>	Read margin choice.
<i>failedAddress</i>	A pointer to the returned failing address.
<i>failedData</i>	A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

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

Function Documentation

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

15.5.18 **status_t FLASH_VerifyEraseAllExecuteOnlySegments (flash_config_t * config, flash_margin_value_t margin)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>margin</i>	Read margin choice.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<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.
-------------------------------------	--

15.5.19 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.

15.5.20 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

Function Documentation

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be checked. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be checked. Must be word-aligned.
<i>access_state</i>	A pointer to the value returned for the current access status code for the desired flash area.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	The parameter is not aligned to the specified baseline.
<i>kStatus_FLASH_AddressError</i>	The address is out of range.

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

Parameters

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

Return values

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

15.5.22 **status_t FLASH_SetFlexramFunction (flash_config_t * config, flash flexram_function_option_t option)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>option</i>	The option used to set the work mode of FlexRAM.

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.

15.5.23 `status_t FLASH_SwapControl (flash_config_t * config, uint32_t address, flash_swap_control_option_t option, flash_swap_state_config_t * returnInfo)`

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>address</i>	Address used to configure the flash Swap function.
<i>option</i>	The possible option used to configure Flash Swap function or check the flash Swap status
<i>returnInfo</i>	A pointer to the data which is used to return the information of flash Swap.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
------------------------------	--------------------------------

Function Documentation

<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_SwapIndicatorAddressError</i>	Swap indicator address 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.

15.5.24 **status_t FLASH_Swap (flash_config_t * config, uint32_t address, flash_swap_function_option_t option)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>address</i>	Address used to configure the flash swap function
<i>option</i>	The possible option used to configure the Flash Swap function or check the flash Swap status.

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_SwapIndicatorAddressError</i>	Swap indicator address is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_SwapSystemNotInUninitialized</i>	Swap system is not in an uninitialized state.

15.5.25 `status_t FLASH_ProgramPartition (flash_config_t * config, flash_partition_flexram_load_option_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)`

Parameters

<i>config</i>	Pointer to storage for the driver runtime state.
<i>option</i>	The option used to set FlexRAM load behavior during reset.
<i>eepromDataSizeCode</i>	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
<i>flexnvmPartitionCode</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.

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.

15.5.26 **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.

15.5.27 **status_t FLASH_PflashGetProtection (flash_config_t * config, pflash_protection_status_t * protectStatus)**

Parameters

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

<i>protectStatus</i>	Protect status returned by the PFlash IP. Each bit is corresponding to the protection of 1/32(64) of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.
----------------------	---

Return values

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

15.5.28 **status_t FLASH_DflashSetProtection (flash_config_t * *config*, uint8_t *protectStatus*)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the DFlash protection register. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_-CommandNotSupported</i>	Flash API is not supported.
<i>kStatus_FLASH_-CommandFailure</i>	Run-time error during command execution.

15.5.29 **status_t FLASH_DflashGetProtection (flash_config_t * *config*, uint8_t * *protectStatus*)**

Function Documentation

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash, and so on. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_-CommandNotSupported</i>	Flash API is not supported.

15.5.30 **status_t FLASH_EepromSetProtection (flash_config_t * config, uint8_t protectStatus)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the EEPROM protection register. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of EEPROM, and so on. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

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

<i>kStatus_FLASH_CommandNotSupported</i>	Flash API is not supported.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.

15.5.31 **status_t FLASH_EepromGetProtection (*flash_config_t * config, uint8_t * protectStatus*)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of the EEPROM. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_CommandNotSupported</i>	Flash API is not supported.

Function Documentation

Chapter 16

FlexBus: External Bus Interface Driver

16.1 Overview

The KSDK provides a peripheral driver for the Crossbar External Bus Interface (FlexBus) block of Kinetis devices.

A multifunction external bus interface is provided on the device with a basic functionality to interface to slave-only devices. It can be directly connected to the following asynchronous or synchronous devices with little or no additional circuitry.

- External ROMs
- Flash memories
- Programmable logic devices
- Other simple target (slave) devices

For asynchronous devices, a simple chip-select based interface can be used. The FlexBus interface has up to six general purpose chip-selects, FB_CS[5:0]. The number of chip selects available depends on the device and its pin configuration.

16.2 FlexBus functional operation

To configure the FlexBus driver, use one of the two ways to configure the `flexbus_config_t` structure.

1. Using the `FLEXBUS_GetDefaultConfig()` function.
2. Set parameters in the `flexbus_config_t` structure.

To initialize and configure the FlexBus driver, call the `FLEXBUS_Init()` function and pass a pointer to the `flexbus_config_t` structure.

To de-initialize the FlexBus driver, call the `FLEXBUS_Deinit()` function.

16.3 Typical use case and example

This example shows how to write/read to external memory (MRAM) by using the FlexBus module.

```
flexbus_config_t flexbusUserConfig;

FLEXBUS_GetDefaultConfig(&flexbusUserConfig); /* Gets the default configuration. */
/* Configure some parameters when using MRAM */
flexbusUserConfig.waitStates          = 2U;           /* Wait 2 states */
flexbusUserConfig.chipBaseAddress    = MRAM_START_ADDRESS; /* MRAM address for using
                                                               FlexBus */
flexbusUserConfig.chipBaseAddressMask = 7U;           /* 512 kilobytes memory
                                                               size */
FLEXBUS_Init(FB, &flexbusUserConfig); /* Initializes and configures the FlexBus module */

/* Do something */

FLEXBUS_Deinit(FB);
```

Typical use case and example

Data Structures

- struct `flexbus_config_t`
Configuration structure that the user needs to set. [More...](#)

Enumerations

- enum `flexbus_port_size_t`{
 `kFLEXBUS_4Bytes` = 0x00U,
 `kFLEXBUS_1Byte` = 0x01U,
 `kFLEXBUS_2Bytes` = 0x02U }
Defines port size for FlexBus peripheral.
- enum `flexbus_write_address_hold_t`{
 `kFLEXBUS_Hold1Cycle` = 0x00U,
 `kFLEXBUS_Hold2Cycles` = 0x01U,
 `kFLEXBUS_Hold3Cycles` = 0x02U,
 `kFLEXBUS_Hold4Cycles` = 0x03U }
Defines number of cycles to hold address and attributes for FlexBus peripheral.
- enum `flexbus_read_address_hold_t`{
 `kFLEXBUS_Hold1Or0Cycles` = 0x00U,
 `kFLEXBUS_Hold2Or1Cycles` = 0x01U,
 `kFLEXBUS_Hold3Or2Cycle` = 0x02U,
 `kFLEXBUS_Hold4Or3Cycle` = 0x03U }
Defines number of cycles to hold address and attributes for FlexBus peripheral.
- enum `flexbus_address_setup_t`{
 `kFLEXBUS_FirstRisingEdge` = 0x00U,
 `kFLEXBUS_SecondRisingEdge` = 0x01U,
 `kFLEXBUS_ThirdRisingEdge` = 0x02U,
 `kFLEXBUS_FourthRisingEdge` = 0x03U }
Address setup for FlexBus peripheral.
- enum `flexbus_bytelane_shift_t`{
 `kFLEXBUS_NotShifted` = 0x00U,
 `kFLEXBUS_Shifted` = 0x01U }
Defines byte-lane shift for FlexBus peripheral.
- enum `flexbus_multiplex_group1_t`{
 `kFLEXBUS_MultiplexGroup1_FB_ALE` = 0x00U,
 `kFLEXBUS_MultiplexGroup1_FB_CS1` = 0x01U,
 `kFLEXBUS_MultiplexGroup1_FB_TS` = 0x02U }
Defines multiplex group1 valid signals.
- enum `flexbus_multiplex_group2_t`{
 `kFLEXBUS_MultiplexGroup2_FB_CS4` = 0x00U,
 `kFLEXBUS_MultiplexGroup2_FB_TSIZ0` = 0x01U,
 `kFLEXBUS_MultiplexGroup2_FB_BE_31_24` = 0x02U }
Defines multiplex group2 valid signals.
- enum `flexbus_multiplex_group3_t`{
 `kFLEXBUS_MultiplexGroup3_FB_CS5` = 0x00U,
 `kFLEXBUS_MultiplexGroup3_FB_TSIZ1` = 0x01U,
 `kFLEXBUS_MultiplexGroup3_FB_BE_23_16` = 0x02U }

- *Defines multiplex group3 valid signals.*
enum **flexbus_multiplex_group4_t** {
 kFLEXBUS_MultiplexGroup4_FB_TBST = 0x00U,
 kFLEXBUS_MultiplexGroup4_FB_CS2 = 0x01U,
 kFLEXBUS_MultiplexGroup4_FB_BE_15_8 = 0x02U }
- *Defines multiplex group4 valid signals.*
enum **flexbus_multiplex_group5_t** {
 kFLEXBUS_MultiplexGroup5_FB_TA = 0x00U,
 kFLEXBUS_MultiplexGroup5_FB_CS3 = 0x01U,
 kFLEXBUS_MultiplexGroup5_FB_BE_7_0 = 0x02U }
- *Defines multiplex group5 valid signals.*

Driver version

- #define **FSL_FLEXBUS_DRIVER_VERSION** (MAKE_VERSION(2, 0, 1))
Version 2.0.1.

FlexBus functional operation

- void **FLEXBUS_Init** (FB_Type *base, const **flexbus_config_t** *config)
Initializes and configures the FlexBus module.
- void **FLEXBUS_Deinit** (FB_Type *base)
De-initializes a FlexBus instance.
- void **FLEXBUS_GetDefaultConfig** (**flexbus_config_t** *config)
Initializes the FlexBus configuration structure.

16.4 Data Structure Documentation

16.4.1 struct **flexbus_config_t**

Data Fields

- uint8_t **chip**
Chip FlexBus for validation.
- uint8_t **waitStates**
Value of wait states.
- uint32_t **chipBaseAddress**
Chip base address for using FlexBus.
- uint32_t **chipBaseAddressMask**
Chip base address mask.
- bool **writeProtect**
Write protected.
- bool **burstWrite**
Burst-Write enable.
- bool **burstRead**
Burst-Read enable.
- bool **byteEnableMode**
Byte-enable mode support.
- bool **autoAcknowledge**

Enumeration Type Documentation

- `bool extendTransferAddress`
Auto acknowledge setting.
- `bool secondaryWaitStates`
Extend transfer start/extend address latch enable.
- `flexbus_port_size_t portSize`
Secondary wait states number.
- `flexbus_bytelane_shift_t byteLaneShift`
Port size of transfer.
- `flexbus_write_address_hold_t writeAddressHold`
Byte-lane shift enable.
- `flexbus_read_address_hold_t readAddressHold`
Write address hold or deselect option.
- `flexbus_address_setup_t addressSetup`
Read address hold or deselect option.
- `flexbus_multiplex_group1_t group1MultiplexControl`
Address setup setting.
- `flexbus_multiplex_group2_t group2MultiplexControl`
FlexBus Signal Group 1 Multiplex control.
- `flexbus_multiplex_group3_t group3MultiplexControl`
FlexBus Signal Group 2 Multiplex control.
- `flexbus_multiplex_group4_t group4MultiplexControl`
FlexBus Signal Group 3 Multiplex control.
- `flexbus_multiplex_group5_t group5MultiplexControl`
FlexBus Signal Group 4 Multiplex control.
- `flexbus_multiplex_group6_t group6MultiplexControl`
FlexBus Signal Group 5 Multiplex control.

16.5 Macro Definition Documentation

16.5.1 `#define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

16.6 Enumeration Type Documentation

16.6.1 `enum flexbus_port_size_t`

Enumerator

`kFLEXBUS_4Bytes` 32-bit port size

`kFLEXBUS_1Byte` 8-bit port size

`kFLEXBUS_2Bytes` 16-bit port size

16.6.2 `enum flexbus_write_address_hold_t`

Enumerator

`kFLEXBUS_Hold1Cycle` Hold address and attributes one cycles after FB_CSn negates on writes.

`kFLEXBUS_Hold2Cycles` Hold address and attributes two cycles after FB_CSn negates on writes.

`kFLEXBUS_Hold3Cycles` Hold address and attributes three cycles after FB_CSn negates on writes.

kFLEXBUS_Hold4Cycles Hold address and attributes four cycles after FB_CSn negates on writes.

16.6.3 enum flexbus_read_address_hold_t

Enumerator

kFLEXBUS_Hold1Or0Cycles Hold address and attributes 1 or 0 cycles on reads.

kFLEXBUS_Hold2Or1Cycles Hold address and attributes 2 or 1 cycles on reads.

kFLEXBUS_Hold3Or2Cycle Hold address and attributes 3 or 2 cycles on reads.

kFLEXBUS_Hold4Or3Cycle Hold address and attributes 4 or 3 cycles on reads.

16.6.4 enum flexbus_address_setup_t

Enumerator

kFLEXBUS_FirstRisingEdge Assert FB_CSn on first rising clock edge after address is asserted.

kFLEXBUS_SecondRisingEdge Assert FB_CSn on second rising clock edge after address is asserted.

kFLEXBUS_ThirdRisingEdge Assert FB_CSn on third rising clock edge after address is asserted.

kFLEXBUS_FourthRisingEdge Assert FB_CSn on fourth rising clock edge after address is asserted.

16.6.5 enum flexbus_bytelane_shift_t

Enumerator

kFLEXBUS_NotShifted Not shifted. Data is left-justified on FB_AD

kFLEXBUS_Shifted Shifted. Data is right justified on FB_AD

16.6.6 enum flexbus_multiplex_group1_t

Enumerator

kFLEXBUS_MultiplexGroup1_FB_ALE FB_ALE.

kFLEXBUS_MultiplexGroup1_FB_CS1 FB_CS1.

kFLEXBUS_MultiplexGroup1_FB_TS FB_TS.

Function Documentation

16.6.7 enum flexbus_multiplex_group2_t

Enumerator

kFLEXBUS_MultiplexGroup2_FB_CS4 FB_CS4.
kFLEXBUS_MultiplexGroup2_FB_TSIZ0 FB_TSIZ0.
kFLEXBUS_MultiplexGroup2_FB_BE_31_24 FB_BE_31_24.

16.6.8 enum flexbus_multiplex_group3_t

Enumerator

kFLEXBUS_MultiplexGroup3_FB_CS5 FB_CS5.
kFLEXBUS_MultiplexGroup3_FB_TSIZ1 FB_TSIZ1.
kFLEXBUS_MultiplexGroup3_FB_BE_23_16 FB_BE_23_16.

16.6.9 enum flexbus_multiplex_group4_t

Enumerator

kFLEXBUS_MultiplexGroup4_FB_TBST FB_TBST.
kFLEXBUS_MultiplexGroup4_FB_CS2 FB_CS2.
kFLEXBUS_MultiplexGroup4_FB_BE_15_8 FB_BE_15_8.

16.6.10 enum flexbus_multiplex_group5_t

Enumerator

kFLEXBUS_MultiplexGroup5_FB_TA FB_TA.
kFLEXBUS_MultiplexGroup5_FB_CS3 FB_CS3.
kFLEXBUS_MultiplexGroup5_FB_BE_7_0 FB_BE_7_0.

16.7 Function Documentation

16.7.1 void FLEXBUS_Init (FB_Type * *base*, const flexbus_config_t * *config*)

This function enables the clock gate for FlexBus module. Only chip 0 is validated and set to known values. Other chips are disabled. Note that in this function, certain parameters, depending on external memories, must be set before using the [FLEXBUS_Init\(\)](#) function. This example shows how to set up the *uart_state_t* and the *flexbus_config_t* parameters and how to call the FLEXBUS_Init function by passing in these parameters.

```

flexbus_config_t flexbusConfig;
FLEXBUS_GetDefaultConfig(&flexbusConfig);
flexbusConfig.waitStates          = 2U;
flexbusConfig.chipBaseAddress     = 0x60000000U;
flexbusConfig.chipBaseAddressMask = 7U;
FLEXBUS_Init(FB, &flexbusConfig);

```

Parameters

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

16.7.2 void FLEXBUS_Deinit (FB_Type * *base*)

This function disables the clock gate of the FlexBus module clock.

Parameters

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

16.7.3 void FLEXBUS_GetDefaultConfig (flexbus_config_t * *config*)

This function initializes the FlexBus configuration structure to default value. The default values are.

```

fbConfig->chip                  = 0;
fbConfig->writeProtect          = 0;
fbConfig->burstWrite            = 0;
fbConfig->burstRead             = 0;
fbConfig->byteEnableMode        = 0;
fbConfig->autoAcknowledge       = true;
fbConfig->extendTransferAddress = 0;
fbConfig->secondaryWaitStates   = 0;
fbConfig->byteLaneShift         = kFLEXBUS_NotShifted;
fbConfig->writeAddressHold      = kFLEXBUS_Hold1Cycle;
fbConfig->readAddressHold       = kFLEXBUS_Hold1Or0Cycles;
fbConfig->addressSetup          = kFLEXBUS_FirstRisingEdge;
fbConfig->portSize              = kFLEXBUS_1Byte;
fbConfig->group1MultiplexControl = kFLEXBUS_MultiplexGroup1_FB_ALE;
fbConfig->group2MultiplexControl = kFLEXBUS_MultiplexGroup2_FB_CS4 ;
fbConfig->group3MultiplexControl = kFLEXBUS_MultiplexGroup3_FB_CS5;
fbConfig->group4MultiplexControl = kFLEXBUS_MultiplexGroup4_FB_TBST;
fbConfig->group5MultiplexControl = kFLEXBUS_MultiplexGroup5_FB_TA;

```

Function Documentation

Parameters

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

See Also

[FLEXBUS_Init](#)

Chapter 17

FlexCAN: Flex Controller Area Network Driver

17.1 Overview

The KSDK provides a peripheral driver for the Flex Controller Area Network (FlexCAN) module of Kinetis devices.

Modules

- [FlexCAN Driver](#)
- [FlexCAN eDMA Driver](#)

FlexCAN Driver

17.2 FlexCAN Driver

17.2.1 Overview

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

17.2.2 Typical use case

17.2.2.1 Message Buffer Send Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t txFrame;

/* Init FlexCAN module. */
FLEXCAN_SetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enable FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the transmit message buffer. */
FLEXCAN_SetTxMbConfig(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, true);

/* Prepares the transmit frame for sending. */
txFrame.format = KFLEXCAN_FrameFormatStandard;
txFrame.type   = KFLEXCAN_FrameTypeData;
txFrame.id     = FLEXCAN_ID_STD(0x123);
txFrame.length = 8;
txFrame.dataWord0 = CAN_WORD0_DATA_BYTE_0(0x11) |
                   CAN_WORD0_DATA_BYTE_1(0x22) |
                   CAN_WORD0_DATA_BYTE_2(0x33) |
                   CAN_WORD0_DATA_BYTE_3(0x44);
txFrame.dataWord1 = CAN_WORD1_DATA_BYTE_4(0x55) |
                   CAN_WORD1_DATA_BYTE_5(0x66) |
                   CAN_WORD1_DATA_BYTE_6(0x77) |
                   CAN_WORD1_DATA_BYTE_7(0x88);
/* Writes a transmit message buffer to send a CAN Message. */
FLEXCAN_WriteTxMb(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, &txFrame);

/* Waits until the transmit message buffer is empty. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX));

/* Cleans the transmit message buffer empty status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX);
```

17.2.2.2 Message Buffer Receive Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module. */
FLEXCAN_SetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive message buffer. */
```

```

mbConfig.format      = KFLEXCAN_FrameFormatStandard;
mbConfig.type        = KFLEXCAN_FrameTypeData;
mbConfig.id          = FLEXCAN_ID_STD(0x123);
FLEXCAN_SetRxMbConfig(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &mbConfig, true);

/* Waits until the receive message buffer is full. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX));

/* Reads the received message from the receive message buffer. */
FLEXCAN_ReadRxMb(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &rxFrame);

/* Cleans the receive message buffer full status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX);

```

17.2.2.3 Receive FIFO Operation

```

uint32_t rxFifoFilter[] = {FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 1, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 1, 0)};
;

flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module. */
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive FIFO. */
rxFifoConfig.idFilterTable = rxFifoFilter;
rxFifoConfig.idFilterType  = KFLEXCAN_RxFifoFilterTypeA;
rxFifoConfig.idFilterNum   = sizeof(rxFifoFilter) / sizeof(rxFifoFilter[0]);
rxFifoConfig.priority     = KFLEXCAN_RxFifoPrioHigh;
FLEXCAN_SetRxFifoConfig(EXAMPLE_CAN, &rxFifoConfig, true);

/* Waits until the receive FIFO becomes available. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag));

/* Reads the message from the receive FIFO. */
FLEXCAN_ReadRxFifo(EXAMPLE_CAN, &rxFrame);

/* Cleans the receive FIFO available status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag);

```

Data Structures

- struct [flexcan_frame_t](#)
FlexCAN message frame structure. [More...](#)
- struct [flexcan_config_t](#)
FlexCAN module configuration structure. [More...](#)
- struct [flexcan_timing_config_t](#)
FlexCAN protocol timing characteristic configuration structure. [More...](#)
- struct [flexcan_rx_mb_config_t](#)
FlexCAN Receive Message Buffer configuration structure. [More...](#)
- struct [flexcan_rx_fifo_config_t](#)

FlexCAN Driver

FlexCAN Rx FIFO configuration structure. [More...](#)

- struct **flexcan_mb_transfer_t**
FlexCAN Message Buffer transfer. [More...](#)
- struct **flexcan_fifo_transfer_t**
FlexCAN Rx FIFO transfer. [More...](#)
- struct **flexcan_handle_t**
FlexCAN handle structure. [More...](#)

Macros

- #define **FLEXCAN_ID_STD**(id) (((uint32_t)((uint32_t)(id)) << CAN_ID_STD_SHIFT)) & CAN_ID_STD_MASK)
FlexCAN Frame ID helper macro.
- #define **FLEXCAN_ID_EXT**(id)
Extend Frame ID helper macro.
- #define **FLEXCAN_RX_MB_STD_MASK**(id, rtr, ide)
FlexCAN Rx Message Buffer Mask helper macro.
- #define **FLEXCAN_RX_MB_EXT_MASK**(id, rtr, ide)
Extend Rx Message Buffer Mask helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_A**(id, rtr, ide)
FlexCAN Rx FIFO Mask helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH**(id, rtr, ide)
Standard Rx FIFO Mask helper macro Type B upper part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW**(id, rtr, ide)
Standard Rx FIFO Mask helper macro Type B lower part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH**(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 21)
Standard Rx FIFO Mask helper macro Type C upper part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH**(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 13)
Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW**(id) ((FLEXCAN_ID_STD(id) & 0x7F8) << 5)
Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW**(id) ((FLEXCAN_ID_STD(id) & 0x7F8) >> 3)
Standard Rx FIFO Mask helper macro Type C lower part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A**(id, rtr, ide)
Extend Rx FIFO Mask helper macro Type A helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH**(id, rtr, ide)
Extend Rx FIFO Mask helper macro Type B upper part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW**(id, rtr, ide)
Extend Rx FIFO Mask helper macro Type B lower part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH**(id) ((FLEXCAN_ID_EXT(id) & 0x1FE00000) << 3)
Extend Rx FIFO Mask helper macro Type C upper part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH**(id)
Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW**(id)

- Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW**(id) ((**FLEXCAN_ID_EXT**(id) & 0x1FE00000) >> 21)
 - Extend Rx FIFO Mask helper macro Type C lower part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A**(id, rtr, ide) **FLEXCAN_RX_FIFO_STD-MASK_TYPE_A**(id, rtr, ide)
 - FlexCAN Rx FIFO Filter helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH**(id, rtr, ide)
 - Standard Rx FIFO Filter helper macro Type B upper part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW**(id, rtr, ide)
 - Standard Rx FIFO Filter helper macro Type B lower part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH**(id)
 - Standard Rx FIFO Filter helper macro Type C upper part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH**(id)
 - Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW**(id)
 - Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.
- #define **FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW**(id) **FLEXCAN_RX_FIFO_STD-MASK_TYPE_C_LOW**(id)
 - Standard Rx FIFO Filter helper macro Type C lower part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A**(id, rtr, ide) **FLEXCAN_RX_FIFO_EXT-MASK_TYPE_A**(id, rtr, ide)
 - Extend Rx FIFO Filter helper macro Type A helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH**(id, rtr, ide)
 - Extend Rx FIFO Filter helper macro Type B upper part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW**(id, rtr, ide)
 - Extend Rx FIFO Filter helper macro Type B lower part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH**(id) **FLEXCAN_RX_FIFO_EXT-MASK_TYPE_C_HIGH**(id)
 - Extend Rx FIFO Filter helper macro Type C upper part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH**(id)
 - Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW**(id)
 - Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.
- #define **FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW**(id) **FLEXCAN_RX_FIFO_EXT-MASK_TYPE_C_LOW**(id)
 - Extend Rx FIFO Filter helper macro Type C lower part helper macro.

Typedefs

- typedef void(* **flexcan_transfer_callback_t**)(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint32_t result, void *userData)
 - FlexCAN transfer callback function.*

Enumerations

- enum `_flexcan_status` {
 kStatus_FLEXCAN_TxBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 0),
 kStatus_FLEXCAN_TxIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 1),
 kStatus_FLEXCAN_TxSwitchToRx,
 kStatus_FLEXCAN_RxBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 3),
 kStatus_FLEXCAN_RxIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 4),
 kStatus_FLEXCAN_RxOverflow = MAKE_STATUS(kStatusGroup_FLEXCAN, 5),
 kStatus_FLEXCAN_RxFifoBusy = MAKE_STATUS(kStatusGroup_FLEXCAN, 6),
 kStatus_FLEXCAN_RxFifoIdle = MAKE_STATUS(kStatusGroup_FLEXCAN, 7),
 kStatus_FLEXCAN_RxFifoOverflow = MAKE_STATUS(kStatusGroup_FLEXCAN, 8),
 kStatus_FLEXCAN_RxFifoWarning = MAKE_STATUS(kStatusGroup_FLEXCAN, 0),
 kStatus_FLEXCAN_ErrorStatus = MAKE_STATUS(kStatusGroup_FLEXCAN, 10),
 kStatus_FLEXCAN_UnHandled = MAKE_STATUS(kStatusGroup_FLEXCAN, 11) }
 FlexCAN transfer status.
- enum `flexcan_frame_format_t` {
 kFLEXCAN_FrameFormatStandard = 0x0U,
 kFLEXCAN_FrameFormatExtend = 0x1U }
 FlexCAN frame format.
- enum `flexcan_frame_type_t` {
 kFLEXCAN_FrameTypeData = 0x0U,
 kFLEXCAN_FrameTypeRemote = 0x1U }
 FlexCAN frame type.
- enum `flexcan_clock_source_t` {
 kFLEXCAN_ClkSrcOsc = 0x0U,
 kFLEXCAN_ClkSrcPeri = 0x1U }
 FlexCAN clock source.
- enum `flexcan_rx_fifo_filter_type_t` {
 kFLEXCAN_RxFifoFilterTypeA = 0x0U,
 kFLEXCAN_RxFifoFilterTypeB,
 kFLEXCAN_RxFifoFilterTypeC,
 kFLEXCAN_RxFifoFilterTypeD = 0x3U }
 FlexCAN Rx Fifo Filter type.
- enum `flexcan_rx_fifo_priority_t` {
 kFLEXCAN_RxFifoPrioLow = 0x0U,
 kFLEXCAN_RxFifoPrioHigh = 0x1U }
 FlexCAN Rx FIFO priority.
- enum `_flexcan_interrupt_enable` {
 kFLEXCAN_BusOffInterruptEnable = CAN_CTRL1_BOFFMSK_MASK,
 kFLEXCAN_ErrorInterruptEnable = CAN_CTRL1_ERRMSK_MASK,
 kFLEXCAN_RxWarningInterruptEnable = CAN_CTRL1_RWRNMSK_MASK,
 kFLEXCAN_TxWarningInterruptEnable = CAN_CTRL1_TWRNMSK_MASK,
 kFLEXCAN_WakeUpInterruptEnable = CAN_MCR_WAKMSK_MASK }
 FlexCAN interrupt configuration structure, default settings all disabled.
- enum `_flexcan_flags` {

```

kFLEXCAN_SynchFlag = CAN_ESR1_SYNCH_MASK,
kFLEXCAN_TxWarningIntFlag = CAN_ESR1_TWRNINT_MASK,
kFLEXCAN_RxWarningIntFlag = CAN_ESR1_RWRNINT_MASK,
kFLEXCAN_TxErrorWarningFlag = CAN_ESR1_TXWRN_MASK,
kFLEXCAN_RxErrorWarningFlag = CAN_ESR1_RXWRN_MASK,
kFLEXCAN_IdleFlag = CAN_ESR1_IDLE_MASK,
kFLEXCAN_FaultConfinementFlag = CAN_ESR1_FLTCONF_MASK,
kFLEXCAN_TransmittingFlag = CAN_ESR1_TX_MASK,
kFLEXCAN_ReceivingFlag = CAN_ESR1_RX_MASK,
kFLEXCAN_BusOffIntFlag = CAN_ESR1_BOFFINT_MASK,
kFLEXCAN_ErrorIntFlag = CAN_ESR1_ERRINT_MASK,
kFLEXCAN_WakeUpIntFlag = CAN_ESR1_WAKINT_MASK,
kFLEXCAN_ErrorFlag }

```

FlexCAN status flags.

- enum `_flexcan_error_flags` {

```

kFLEXCAN_StuffingError = CAN_ESR1_STFERR_MASK,
kFLEXCAN_FormError = CAN_ESR1_FRMERR_MASK,
kFLEXCAN_CrcError = CAN_ESR1_CRCERR_MASK,
kFLEXCAN_AckError = CAN_ESR1_ACKERR_MASK,
kFLEXCAN_Bit0Error = CAN_ESR1_BIT0ERR_MASK,
kFLEXCAN_Bit1Error = CAN_ESR1_BIT1ERR_MASK }

```

FlexCAN error status flags.

- enum `_flexcan_rx_fifo_flags` {

```

kFLEXCAN_RxFifoOverflowFlag = CAN_IFLAG1_BUF7I_MASK,
kFLEXCAN_RxFifoWarningFlag = CAN_IFLAG1_BUF6I_MASK,
kFLEXCAN_RxFifoFrameAvlFlag = CAN_IFLAG1_BUF5I_MASK }

```

FlexCAN Rx FIFO status flags.

Driver version

- #define `FLEXCAN_DRIVER_VERSION` (MAKE_VERSION(2, 1, 0))
FlexCAN driver version 2.1.0.

Initialization and deinitialization

- void `FLEXCAN_Init` (CAN_Type *base, const `flexcan_config_t` *config, uint32_t sourceClock_Hz)
Initializes a FlexCAN instance.
- void `FLEXCAN_Deinit` (CAN_Type *base)
De-initializes a FlexCAN instance.
- void `FLEXCAN_GetDefaultConfig` (`flexcan_config_t` *config)
Gets the default configuration structure.

FlexCAN Driver

Configuration.

- void [FLEXCAN_SetTimingConfig](#) (CAN_Type *base, const [flexcan_timing_config_t](#) *config)
Sets the FlexCAN protocol timing characteristic.
- void [FLEXCAN_SetRxMbGlobalMask](#) (CAN_Type *base, uint32_t mask)
Sets the FlexCAN receive message buffer global mask.
- void [FLEXCAN_SetRxFifoGlobalMask](#) (CAN_Type *base, uint32_t mask)
Sets the FlexCAN receive FIFO global mask.
- void [FLEXCAN_SetRxIndividualMask](#) (CAN_Type *base, uint8_t maskIdx, uint32_t mask)
Sets the FlexCAN receive individual mask.
- void [FLEXCAN_SetTxMbConfig](#) (CAN_Type *base, uint8_t mbIdx, bool enable)
Configures a FlexCAN transmit message buffer.
- void [FLEXCAN_SetRxMbConfig](#) (CAN_Type *base, uint8_t mbIdx, const [flexcan_rx_mb_config_t](#) *config, bool enable)
Configures a FlexCAN Receive Message Buffer.
- void [FLEXCAN_SetRxFifoConfig](#) (CAN_Type *base, const [flexcan_rx_fifo_config_t](#) *config, bool enable)
Configures the FlexCAN Rx FIFO.

Status

- static uint32_t [FLEXCAN_GetStatusFlags](#) (CAN_Type *base)
Gets the FlexCAN module interrupt flags.
- static void [FLEXCAN_ClearStatusFlags](#) (CAN_Type *base, uint32_t mask)
Clears status flags with the provided mask.
- static void [FLEXCAN_GetBusErrCount](#) (CAN_Type *base, uint8_t *txErrBuf, uint8_t *rxErrBuf)
Gets the FlexCAN Bus Error Counter value.
- static uint32_t [FLEXCAN_GetMbStatusFlags](#) (CAN_Type *base, uint32_t mask)
Gets the FlexCAN Message Buffer interrupt flags.
- static void [FLEXCAN_ClearMbStatusFlags](#) (CAN_Type *base, uint32_t mask)
Clears the FlexCAN Message Buffer interrupt flags.

Interrupts

- static void [FLEXCAN_EnableInterrupts](#) (CAN_Type *base, uint32_t mask)
Enables FlexCAN interrupts according to the provided mask.
- static void [FLEXCAN_DisableInterrupts](#) (CAN_Type *base, uint32_t mask)
Disables FlexCAN interrupts according to the provided mask.
- static void [FLEXCAN_EnableMbInterrupts](#) (CAN_Type *base, uint32_t mask)
Enables FlexCAN Message Buffer interrupts.
- static void [FLEXCAN_DisableMbInterrupts](#) (CAN_Type *base, uint32_t mask)
Disables FlexCAN Message Buffer interrupts.

Bus Operations

- static void [FLEXCAN_Enable](#) (CAN_Type *base, bool enable)
Enables or disables the FlexCAN module operation.

- status_t **FLEXCAN_WriteTxMb** (CAN_Type *base, uint8_t mbIdx, const flexcan_frame_t *txFrame)
Writes a FlexCAN Message to the Transmit Message Buffer.
- status_t **FLEXCAN_ReadRxMb** (CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *rxFrame)
Reads a FlexCAN Message from Receive Message Buffer.
- status_t **FLEXCAN_ReadRxFifo** (CAN_Type *base, flexcan_frame_t *rxFrame)
Reads a FlexCAN Message from Rx FIFO.

Transactional

- status_t **FLEXCAN_TransferSendBlocking** (CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *txFrame)
Performs a polling send transaction on the CAN bus.
- status_t **FLEXCAN_TransferReceiveBlocking** (CAN_Type *base, uint8_t mbIdx, flexcan_frame_t *rxFrame)
Performs a polling receive transaction on the CAN bus.
- status_t **FLEXCAN_TransferReceiveFifoBlocking** (CAN_Type *base, flexcan_frame_t *rxFrame)
Performs a polling receive transaction from Rx FIFO on the CAN bus.
- void **FLEXCAN_TransferCreateHandle** (CAN_Type *base, flexcan_handle_t *handle, flexcan_transfer_callback_t callback, void *userData)
Initializes the FlexCAN handle.
- status_t **FLEXCAN_TransferSendNonBlocking** (CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *xfer)
Sends a message using IRQ.
- status_t **FLEXCAN_TransferReceiveNonBlocking** (CAN_Type *base, flexcan_handle_t *handle, flexcan_mb_transfer_t *xfer)
Receives a message using IRQ.
- status_t **FLEXCAN_TransferReceiveFifoNonBlocking** (CAN_Type *base, flexcan_handle_t *handle, flexcan_fifo_transfer_t *xfer)
Receives a message from Rx FIFO using IRQ.
- void **FLEXCAN_TransferAbortSend** (CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)
Aborts the interrupt driven message send process.
- void **FLEXCAN_TransferAbortReceive** (CAN_Type *base, flexcan_handle_t *handle, uint8_t mbIdx)
Aborts the interrupt driven message receive process.
- void **FLEXCAN_TransferAbortReceiveFifo** (CAN_Type *base, flexcan_handle_t *handle)
Aborts the interrupt driven message receive from Rx FIFO process.
- void **FLEXCAN_TransferHandleIRQ** (CAN_Type *base, flexcan_handle_t *handle)
FlexCAN IRQ handle function.

17.2.3 Data Structure Documentation

17.2.3.1 struct flexcan_frame_t

17.2.3.1.0.46 Field Documentation

17.2.3.1.0.46.1 uint32_t flexcan_frame_t::timestamp

17.2.3.1.0.46.2 uint32_t flexcan_frame_t::length

17.2.3.1.0.46.3 uint32_t flexcan_frame_t::type

17.2.3.1.0.46.4 uint32_t flexcan_frame_t::format

17.2.3.1.0.46.5 uint32_t flexcan_frame_t::__pad0__

17.2.3.1.0.46.6 uint32_t flexcan_frame_t::idhit

17.2.3.1.0.46.7 uint32_t flexcan_frame_t::id

17.2.3.1.0.46.8 uint32_t flexcan_frame_t::dataWord0

17.2.3.1.0.46.9 uint32_t flexcan_frame_t::dataWord1

17.2.3.1.0.46.10 uint8_t flexcan_frame_t::dataByte3

17.2.3.1.0.46.11 uint8_t flexcan_frame_t::dataByte2

17.2.3.1.0.46.12 uint8_t flexcan_frame_t::dataByte1

17.2.3.1.0.46.13 uint8_t flexcan_frame_t::dataByte0

17.2.3.1.0.46.14 uint8_t flexcan_frame_t::dataByte7

17.2.3.1.0.46.15 uint8_t flexcan_frame_t::dataByte6

17.2.3.1.0.46.16 uint8_t flexcan_frame_t::dataByte5

17.2.3.1.0.46.17 uint8_t flexcan_frame_t::dataByte4

17.2.3.2 struct flexcan_config_t

Data Fields

- uint32_t baudRate
FlexCAN baud rate in bps.
- flexcan_clock_source_t clkSrc
Clock source for FlexCAN Protocol Engine.
- uint8_t maxMbNum
The maximum number of Message Buffers used by user

- bool `enableLoopBack`
Enable or Disable Loop Back Self Test Mode.
- bool `enableSelfWakeup`
Enable or Disable Self Wakeup Mode.
- bool `enableIndividMask`
Enable or Disable Rx Individual Mask.

17.2.3.2.0.47 Field Documentation

17.2.3.2.0.47.1 `uint32_t flexcan_config_t::baudRate`

17.2.3.2.0.47.2 `flexcan_clock_source_t flexcan_config_t::clkSrc`

17.2.3.2.0.47.3 `uint8_t flexcan_config_t::maxMbNum`

17.2.3.2.0.47.4 `bool flexcan_config_t::enableLoopBack`

17.2.3.2.0.47.5 `bool flexcan_config_t::enableSelfWakeup`

17.2.3.2.0.47.6 `bool flexcan_config_t::enableIndividMask`

17.2.3.3 `struct flexcan_timing_config_t`

Data Fields

- `uint8_t preDivider`
Clock Pre-scaler Division Factor.
- `uint8_t rJumpwidth`
Re-sync Jump Width.
- `uint8_t phaseSeg1`
Phase Segment 1.
- `uint8_t phaseSeg2`
Phase Segment 2.
- `uint8_t propSeg`
Propagation Segment.

FlexCAN Driver

17.2.3.3.0.48 Field Documentation

17.2.3.3.0.48.1 `uint8_t flexcan_timing_config_t::preDivider`

17.2.3.3.0.48.2 `uint8_t flexcan_timing_config_t::rJumpwidth`

17.2.3.3.0.48.3 `uint8_t flexcan_timing_config_t::phaseSeg1`

17.2.3.3.0.48.4 `uint8_t flexcan_timing_config_t::phaseSeg2`

17.2.3.3.0.48.5 `uint8_t flexcan_timing_config_t::propSeg`

17.2.3.4 `struct flexcan_rx_mb_config_t`

This structure is used as the parameter of [FLEXCAN_SetRxMbConfig\(\)](#) function. The [FLEXCAN_SetRxMbConfig\(\)](#) function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

Data Fields

- `uint32_t id`
CAN Message Buffer Frame Identifier, should be set using [FLEXCAN_ID_EXT\(\)](#) or [FLEXCAN_ID_STD\(\)](#) macro.
- `flexcan_frame_format_t format`
CAN Frame Identifier format(Standard or Extend).
- `flexcan_frame_type_t type`
CAN Frame Type(Data or Remote).

17.2.3.4.0.49 Field Documentation

17.2.3.4.0.49.1 `uint32_t flexcan_rx_mb_config_t::id`

17.2.3.4.0.49.2 `flexcan_frame_format_t flexcan_rx_mb_config_t::format`

17.2.3.4.0.49.3 `flexcan_frame_type_t flexcan_rx_mb_config_t::type`

17.2.3.5 `struct flexcan_rx_fifo_config_t`

Data Fields

- `uint32_t * idFilterTable`
Pointer to the FlexCAN Rx FIFO identifier filter table.
- `uint8_t idFilterNum`
The quantity of filter elements.
- `flexcan_rx_fifo_filter_type_t idFilterType`
The FlexCAN Rx FIFO Filter type.
- `flexcan_rx_fifo_priority_t priority`
The FlexCAN Rx FIFO receive priority.

17.2.3.5.0.50 Field Documentation**17.2.3.5.0.50.1 uint32_t* flexcan_rx_fifo_config_t::idFilterTable****17.2.3.5.0.50.2 uint8_t flexcan_rx_fifo_config_t::idFilterNum****17.2.3.5.0.50.3 flexcan_rx_fifo_filter_type_t flexcan_rx_fifo_config_t::idFilterType****17.2.3.5.0.50.4 flexcan_rx_fifo_priority_t flexcan_rx_fifo_config_t::priority****17.2.3.6 struct flexcan_mb_transfer_t****Data Fields**

- **flexcan_frame_t * frame**
The buffer of CAN Message to be transfer.
- **uint8_t mbIdx**
The index of Message buffer used to transfer Message.

17.2.3.6.0.51 Field Documentation**17.2.3.6.0.51.1 flexcan_frame_t* flexcan_mb_transfer_t::frame****17.2.3.6.0.51.2 uint8_t flexcan_mb_transfer_t::mbIdx****17.2.3.7 struct flexcan_fifo_transfer_t****Data Fields**

- **flexcan_frame_t * frame**
The buffer of CAN Message to be received from Rx FIFO.

17.2.3.7.0.52 Field Documentation**17.2.3.7.0.52.1 flexcan_frame_t* flexcan_fifo_transfer_t::frame****17.2.3.8 struct _flexcan_handle**

FlexCAN handle structure definition.

Data Fields

- **flexcan_transfer_callback_t callback**
Callback function.
- **void * userData**
FlexCAN callback function parameter.
- **flexcan_frame_t *volatile mbFrameBuf [CAN_WORD1_COUNT]**
The buffer for received data from Message Buffers.
- **flexcan_frame_t *volatile rxFifoFrameBuf**
The buffer for received data from Rx FIFO.

FlexCAN Driver

- volatile uint8_t **mbState** [CAN_WORD1_COUNT]
Message Buffer transfer state.
- volatile uint8_t **rxFifoState**
Rx FIFO transfer state.

17.2.3.8.0.53 Field Documentation

17.2.3.8.0.53.1 **flexcan_transfer_callback_t flexcan_handle_t::callback**

17.2.3.8.0.53.2 **void* flexcan_handle_t::userData**

17.2.3.8.0.53.3 **flexcan_frame_t* volatile flexcan_handle_t::mbFrameBuf[CAN_WORD1_COUNT]**

17.2.3.8.0.53.4 **flexcan_frame_t* volatile flexcan_handle_t::rx_fifoFrameBuf**

17.2.3.8.0.53.5 **volatile uint8_t flexcan_handle_t::mbState[CAN_WORD1_COUNT]**

17.2.3.8.0.53.6 **volatile uint8_t flexcan_handle_t::rxFifoState**

17.2.4 Macro Definition Documentation

17.2.4.1 **#define FLEXCAN_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))**

17.2.4.2 **#define FLEXCAN_ID_STD(id) (((uint32_t)((uint32_t)(id)) << CAN_ID_STD_SHIFT) & CAN_ID_STD_MASK)**

Standard Frame ID helper macro.

17.2.4.3 **#define FLEXCAN_ID_EXT(id)**

Value:

```
((uint32_t)((uint32_t)(id)) << CAN_ID_EXT_SHIFT) & \  
 (CAN_ID_EXT_MASK | CAN_ID_STD_MASK))
```

17.2.4.4 **#define FLEXCAN_RX_MB_STD_MASK(id, rtr, ide)**

Value:

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \  
 FLEXCAN_ID_STD(id))
```

Standard Rx Message Buffer Mask helper macro.

17.2.4.5 #define FLEXCAN_RX_MB_EXT_MASK(*id*, *rtr*, *ide*)**Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
FLEXCAN_ID_EXT(id))
```

17.2.4.6 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(*id*, *rtr*, *ide*)**Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
(FLEXCAN_ID_STD(id) << 1))
```

Standard Rx FIFO Mask helper macro Type A helper macro.

17.2.4.7 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(*id*, *rtr*, *ide*)**Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
(FLEXCAN_ID_STD(id) << 16))
```

17.2.4.8 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(*id*, *rtr*, *ide*)**Value:**

```
((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \
FLEXCAN_ID_STD(id))
```

**17.2.4.9 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(*id*
)(**(FLEXCAN_ID_STD(id) & 0x7F8) << 21**)****17.2.4.10 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(*id*
)(**(FLEXCAN_ID_STD(id) & 0x7F8) << 13**)**

\

FlexCAN Driver

17.2.4.11 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(*id*)
((FLEXCAN_ID_STD(*id*) & 0x7F8) << 5)

17.2.4.12 #define FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(*id*)
((FLEXCAN_ID_STD(*id*) & 0x7F8) >> 3)

17.2.4.13 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(*id*, *rtr*, *ide*)

Value:

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \  
  (FLEXCAN_ID_EXT(id) << 1))
```

17.2.4.14 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(*id*, *rtr*, *ide*)

Value:

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \  
  ((FLEXCAN_ID_EXT(id) & 0x1FFF8000) << 1))
```

17.2.4.15 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(*id*, *rtr*, *ide*)

Value:

```
((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \  
  ((FLEXCAN_ID_EXT(id) & 0x1FFF8000) >> 15)) \
```

17.2.4.16 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(*id*)
((FLEXCAN_ID_EXT(*id*) & 0x1FE00000) << 3)

17.2.4.17 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(*id*)

Value:

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> 5) \
```

17.2.4.18 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(*id*)

Value:

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> 13) \
```

17.2.4.19 #define FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(*id*)
 $((\text{FLEXCAN_ID_EXT}(\text{id}) \& 0x1FE00000) >> 21)$

17.2.4.20 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(*id*, *rtr*, *ide*)
FLEXCAN_RX_FIFO_STD_MASK_TYPE_A(*id*, *rtr*, *ide*)

Standard Rx FIFO Filter helper macro Type A helper macro.

17.2.4.21 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_HIGH(*id*, *rtr*, *ide*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(           \
    id, rtr, ide)
```

17.2.4.22 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_B_LOW(*id*, *rtr*, *ide*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(           \
    id, rtr, ide)
```

17.2.4.23 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(           \
    id)
```

17.2.4.24 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_HIGH(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(           \
    id)
```

17.2.4.25 #define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_MID_LOW(*id*)

Value:

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(           \
    id)
```

FlexCAN Driver

17.2.4.26 `#define FLEXCAN_RX_FIFO_STD_FILTER_TYPE_C_LOW(id) FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_LOW(id)`

\

17.2.4.27 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A(id, rtr, ide) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)`

17.2.4.28 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH(id, rtr, ide)`

Value:

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(id, rtr, ide) \`

17.2.4.29 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW(id, rtr, ide)`

Value:

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(id, rtr, ide) \`

17.2.4.30 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH(id) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)`

\

17.2.4.31 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH(id)`

Value:

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(id) \`

17.2.4.32 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW(id)`

Value:

`FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(id) \`

```
17.2.4.33 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW( id
) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)
```

17.2.5 Typedef Documentation

17.2.5.1 `typedef void(* flexcan_transfer_callback_t)(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint32_t result, void *userData)`

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus_FLEXCAN_ErrorStatus, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

17.2.6 Enumeration Type Documentation

17.2.6.1 `enum _flexcan_status`

Enumerator

- kStatus_FLEXCAN_TxBusy* Tx Message Buffer is Busy.
- kStatus_FLEXCAN_TxIdle* Tx Message Buffer is Idle.
- kStatus_FLEXCAN_TxSwitchToRx* Remote Message is send out and Message buffer changed to Receive one.
- kStatus_FLEXCAN_RxBusy* Rx Message Buffer is Busy.
- kStatus_FLEXCAN_RxIdle* Rx Message Buffer is Idle.
- kStatus_FLEXCAN_RxOverflow* Rx Message Buffer is Overflowed.
- kStatus_FLEXCAN_RxFifoBusy* Rx Message FIFO is Busy.
- kStatus_FLEXCAN_RxFifoIdle* Rx Message FIFO is Idle.
- kStatus_FLEXCAN_RxFifoOverflow* Rx Message FIFO is overflowed.
- kStatus_FLEXCAN_RxFifoWarning* Rx Message FIFO is almost overflowed.
- kStatus_FLEXCAN_ErrorStatus* FlexCAN Module Error and Status.
- kStatus_FLEXCAN_UnHandled* UnHadled Interrupt asserted.

17.2.6.2 `enum flexcan_frame_format_t`

Enumerator

- kFLEXCAN_FrameFormatStandard* Standard frame format attribute.
- kFLEXCAN_FrameFormatExtend* Extend frame format attribute.

FlexCAN Driver

17.2.6.3 enum flexcan_frame_type_t

Enumerator

kFLEXCAN_FrameTypeData Data frame type attribute.

kFLEXCAN_FrameTypeRemote Remote frame type attribute.

17.2.6.4 enum flexcan_clock_source_t

Enumerator

kFLEXCAN_ClkSrcOsc FlexCAN Protocol Engine clock from Oscillator.

kFLEXCAN_ClkSrcPeri FlexCAN Protocol Engine clock from Peripheral Clock.

17.2.6.5 enum flexcan_rx_fifo_filter_type_t

Enumerator

kFLEXCAN_RxFifoFilterTypeA One full ID (standard and extended) per ID Filter element.

kFLEXCAN_RxFifoFilterTypeB Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.

kFLEXCAN_RxFifoFilterTypeC Four partial 8-bit Standard or extended ID slices per ID Filter Table element.

kFLEXCAN_RxFifoFilterTypeD All frames rejected.

17.2.6.6 enum flexcan_rx_fifo_priority_t

The matching process starts from the Rx MB(or Rx FIFO) with higher priority. If no MB(or Rx FIFO filter) is satisfied, the matching process goes on with the Rx FIFO(or Rx MB) with lower priority.

Enumerator

kFLEXCAN_RxFifoPrioLow Matching process start from Rx Message Buffer first.

kFLEXCAN_RxFifoPrioHigh Matching process start from Rx FIFO first.

17.2.6.7 enum _flexcan_interrupt_enable

This structure contains the settings for all of the FlexCAN Module interrupt configurations. Note: FlexCAN Message Buffers and Rx FIFO have their own interrupts.

Enumerator

kFLEXCAN_BusOffInterruptEnable Bus Off interrupt.

kFLEXCAN_ErrorInterruptEnable Error interrupt.
kFLEXCAN_RxWarningInterruptEnable Rx Warning interrupt.
kFLEXCAN_TxWarningInterruptEnable Tx Warning interrupt.
kFLEXCAN_WakeUpInterruptEnable Wake Up interrupt.

17.2.6.8 enum _flexcan_flags

This provides constants for the FlexCAN status flags for use in the FlexCAN functions. Note: The CPU read action clears FLEXCAN_ErrorFlag, therefore user need to read FLEXCAN_ErrorFlag and distinguish which error is occur using [_flexcan_error_flags](#) enumerations.

Enumerator

kFLEXCAN_SynchFlag CAN Synchronization Status.
kFLEXCAN_TxWarningIntFlag Tx Warning Interrupt Flag.
kFLEXCAN_RxWarningIntFlag Rx Warning Interrupt Flag.
kFLEXCAN_TxErrorWarningFlag Tx Error Warning Status.
kFLEXCAN_RxErrorWarningFlag Rx Error Warning Status.
kFLEXCAN_IdleFlag CAN IDLE Status Flag.
kFLEXCAN_FaultConfinementFlag Fault Confinement State Flag.
kFLEXCAN_TransmittingFlag FlexCAN In Transmission Status.
kFLEXCAN_ReceivingFlag FlexCAN In Reception Status.
kFLEXCAN_BusOffIntFlag Bus Off Interrupt Flag.
kFLEXCAN_ErrorIntFlag Error Interrupt Flag.
kFLEXCAN_WakeUpIntFlag Wake-Up Interrupt Flag.
kFLEXCAN_ErrorFlag All FlexCAN Error Status.

17.2.6.9 enum _flexcan_error_flags

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFLEXCAN_ErrorFlag in [_flexcan_flags](#) enumerations to determine which error is generated.

Enumerator

kFLEXCAN_StuffingError Stuffing Error.
kFLEXCAN_FormError Form Error.
kFLEXCAN_CrcError Cyclic Redundancy Check Error.
kFLEXCAN_AckError Received no ACK on transmission.
kFLEXCAN_Bit0Error Unable to send dominant bit.
kFLEXCAN_Bit1Error Unable to send recessive bit.

FlexCAN Driver

17.2.6.10 enum _flexcan_rx_fifo_flags

The FlexCAN Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occupies more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Enumerator

- kFLEXCAN_RxFifoOverflowFlag* Rx FIFO overflow flag.
- kFLEXCAN_RxFifoWarningFlag* Rx FIFO almost full flag.
- kFLEXCAN_RxFifoFrameAvlFlag* Frames available in Rx FIFO flag.

17.2.7 Function Documentation

17.2.7.1 void FLEXCAN_Init (CAN_Type * *base*, const flexcan_config_t * *config*, uint32_t *sourceClock_Hz*)

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the *flexcan_config_t* parameters and how to call the FLEXCAN_Init function by passing in these parameters.

```
*     flexcanConfig_t flexcanConfig;
*     flexcanConfig.clkSrc          = kFLEXCAN_ClkSrcOsc;
*     flexcanConfig.baudRate        = 125000U;
*     flexcanConfig.maxMbNum        = 16;
*     flexcanConfig.enableLoopBack  = false;
*     flexcanConfig.enableSelfWakeup = false;
*     flexcanConfig.enableIndividMask = false;
*     flexcanConfig.enableDoze      = false;
*     FLEXCAN_Init(CAN0, &flexcanConfig, 8000000UL);
*
```

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>config</i>	Pointer to the user-defined configuration structure.
<i>sourceClock_- Hz</i>	FlexCAN Protocol Engine clock source frequency in Hz.

17.2.7.2 void FLEXCAN_Deinit (CAN_Type * *base*)

This function disables the FlexCAN module clock and sets all register values to the reset value.

Parameters

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

17.2.7.3 void FLEXCAN_GetDefaultConfig (flexcan_config_t * *config*)

This function initializes the FlexCAN configuration structure to default values. The default values are as follows. *flexcanConfig->clkSrc* = KFLEXCAN_ClkSrcOsc; *flexcanConfig->baudRate* = 125000U; *flexcanConfig->maxMbNum* = 16; *flexcanConfig->enableLoopBack* = false; *flexcanConfig->enableSelfWakeup* = false; *flexcanConfig->enableIndividMask* = false; *flexcanConfig->enableDoze* = false;

Parameters

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

17.2.7.4 void FLEXCAN_SetTimingConfig (CAN_Type * *base*, const flexcan_timing_config_t * *config*)

This function gives user settings to CAN bus timing characteristic. The function is for an experienced user. For less experienced users, call the [FLEXCAN_Init\(\)](#) and fill the baud rate field with a desired value. This provides the default timing characteristics to the module.

Note that calling [FLEXCAN_SetTimingConfig\(\)](#) overrides the baud rate set in [FLEXCAN_Init\(\)](#).

Parameters

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

17.2.7.5 void FLEXCAN_SetRxMbGlobalMask (CAN_Type * *base*, uint32_t *mask*)

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the [FLEXCAN_Init\(\)](#).

Parameters

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

FlexCAN Driver

<i>mask</i>	Rx Message Buffer Global Mask value.
-------------	--------------------------------------

17.2.7.6 void FLEXCAN_SetRxFifoGlobalMask (CAN_Type * *base*, uint32_t *mask*)

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	Rx Fifo Global Mask value.

17.2.7.7 void FLEXCAN_SetRxIndividualMask (CAN_Type * *base*, uint8_t *maskIdx*, uint32_t *mask*)

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the [FLEXCAN_Init\(\)](#). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>maskIdx</i>	The Index of individual Mask.
<i>mask</i>	Rx Individual Mask value.

17.2.7.8 void FLEXCAN_SetTxMbConfig (CAN_Type * *base*, uint8_t *mbIdx*, bool *enable*)

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

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

<i>mbIdx</i>	The Message Buffer index.
<i>enable</i>	Enable/disable Tx Message Buffer. <ul style="list-style-type: none"> • true: Enable Tx Message Buffer. • false: Disable Tx Message Buffer.

17.2.7.9 void FLEXCAN_SetRxMbConfig (CAN_Type * *base*, uint8_t *mbIdx*, const flexcan_rx_mb_config_t * *config*, bool *enable*)

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mbIdx</i>	The Message Buffer index.
<i>config</i>	Pointer to the FlexCAN Message Buffer configuration structure.
<i>enable</i>	Enable/disable Rx Message Buffer. <ul style="list-style-type: none"> • true: Enable Rx Message Buffer. • false: Disable Rx Message Buffer.

17.2.7.10 void FLEXCAN_SetRxFifoConfig (CAN_Type * *base*, const flexcan_rx_fifo_config_t * *config*, bool *enable*)

This function configures the Rx FIFO with given Rx FIFO configuration.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>config</i>	Pointer to the FlexCAN Rx FIFO configuration structure.
<i>enable</i>	Enable/disable Rx FIFO. <ul style="list-style-type: none"> • true: Enable Rx FIFO. • false: Disable Rx FIFO.

17.2.7.11 static uint32_t FLEXCAN_GetStatusFlags (CAN_Type * *base*) [inline], [static]

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators [_flexcan_flags](#). To check the specific status, compare the return value with enumerators in [_flexcan-](#)

FlexCAN Driver

flags.

Parameters

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

Returns

FlexCAN status flags which are ORed by the enumerators in the `_flexcan_flags`.

17.2.7.12 static void FLEXCAN_ClearStatusFlags (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The status flags to be cleared, it is logical OR value of <code>_flexcan_flags</code> .

17.2.7.13 static void FLEXCAN_GetBusErrCount (CAN_Type * *base*, uint8_t * *txErrBuf*, uint8_t * *rxErrBuf*) [inline], [static]

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>txErrBuf</i>	Buffer to store Tx Error Counter value.
<i>rxErrBuf</i>	Buffer to store Rx Error Counter value.

17.2.7.14 static uint32_t FLEXCAN_GetMbStatusFlags (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function gets the interrupt flags of a given Message Buffers.

FlexCAN Driver

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

17.2.7.15 static void FLEXCAN_ClearMbStatusFlags (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function clears the interrupt flags of a given Message Buffers.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

17.2.7.16 static void FLEXCAN_EnableInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see [_flexcan_interrupt_enable](#).

Parameters

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

17.2.7.17 static void FLEXCAN_DisableInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see [_flexcan_interrupt_enable](#).

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of _flexcan_interrupt_enable .

17.2.7.18 static void FLEXCAN_EnableMbInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function enables the interrupts of given Message Buffers.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

17.2.7.19 static void FLEXCAN_DisableMbInterrupts (CAN_Type * *base*, uint32_t *mask*) [inline], [static]

This function disables the interrupts of given Message Buffers.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

17.2.7.20 static void FLEXCAN_Enable (CAN_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the FlexCAN module.

Parameters

<i>base</i>	FlexCAN base pointer.
<i>enable</i>	true to enable, false to disable.

17.2.7.21 status_t FLEXCAN_WriteTxMb (CAN_Type * *base*, uint8_t *mbIdx*, const flexcan_frame_t * *txFrame*)

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

FlexCAN Driver

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mbIdx</i>	The FlexCAN Message Buffer index.
<i>txFrame</i>	Pointer to CAN message frame to be sent.

Return values

<i>kStatus_Success</i>	- Write Tx Message Buffer Successfully.
<i>kStatus_Fail</i>	- Tx Message Buffer is currently in use.

17.2.7.22 status_t FLEXCAN_ReadRxMb (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *rxFrame*)

This function reads a CAN message from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mbIdx</i>	The FlexCAN Message Buffer index.
<i>rxFrame</i>	Pointer to CAN message frame structure for reception.

Return values

<i>kStatus_Success</i>	- Rx Message Buffer is full and has been read successfully.
<i>kStatus_FLEXCAN_Overflow</i>	- Rx Message Buffer is already overflowed and has been read successfully.
<i>kStatus_Fail</i>	- Rx Message Buffer is empty.

17.2.7.23 status_t FLEXCAN_ReadRxFifo (CAN_Type * *base*, flexcan_frame_t * *rxFrame*)

This function reads a CAN message from the FlexCAN build-in Rx FIFO.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>rxFrame</i>	Pointer to CAN message frame structure for reception.

Return values

<i>kStatus_Success</i>	- Read Message from Rx FIFO successfully.
<i>kStatus_Fail</i>	- Rx FIFO is not enabled.

17.2.7.24 status_t FLEXCAN_TransferSendBlocking (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *txFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

<i>base</i>	FlexCAN peripheral base pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.
<i>txFrame</i>	Pointer to CAN message frame to be sent.

Return values

<i>kStatus_Success</i>	- Write Tx Message Buffer Successfully.
<i>kStatus_Fail</i>	- Tx Message Buffer is currently in use.

17.2.7.25 status_t FLEXCAN_TransferReceiveBlocking (CAN_Type * *base*, uint8_t *mbIdx*, flexcan_frame_t * *rxFrame*)

Note that a transfer handle does not need to be created before calling this API.

Parameters

<i>base</i>	FlexCAN peripheral base pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.

FlexCAN Driver

<i>rxFrame</i>	Pointer to CAN message frame structure for reception.
----------------	---

Return values

<i>kStatus_Success</i>	- Rx Message Buffer is full and has been read successfully.
<i>kStatus_FLEXCAN_Rx_Overflow</i>	- Rx Message Buffer is already overflowed and has been read successfully.
<i>kStatus_Fail</i>	- Rx Message Buffer is empty.

17.2.7.26 **status_t FLEXCAN_TransferReceiveFifoBlocking (CAN_Type * *base*, flexcan_frame_t * *rxFrame*)**

Note that a transfer handle does not need to be created before calling this API.

Parameters

<i>base</i>	FlexCAN peripheral base pointer.
<i>rxFrame</i>	Pointer to CAN message frame structure for reception.

Return values

<i>kStatus_Success</i>	- Read Message from Rx FIFO successfully.
<i>kStatus_Fail</i>	- Rx FIFO is not enabled.

17.2.7.27 **void FLEXCAN_TransferCreateHandle (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_transfer_callback_t *callback*, void * *userData*)**

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

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>callback</i>	The callback function.
<i>userData</i>	The parameter of the callback function.

17.2.7.28 status_t FLEXCAN_TransferSendNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_mb_transfer_t * *xfer*)

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

FlexCAN Driver

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>xfer</i>	FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t .

Return values

<i>kStatus_Success</i>	Start Tx Message Buffer sending process successfully.
<i>kStatus_Fail</i>	Write Tx Message Buffer failed.
<i>kStatus_FLEXCAN_Tx-Busy</i>	Tx Message Buffer is in use.

17.2.7.29 status_t FLEXCAN_TransferReceiveNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_mb_transfer_t * *xfer*)

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>xfer</i>	FlexCAN Message Buffer transfer structure. See the flexcan_mb_transfer_t .

Return values

<i>kStatus_Success</i>	- Start Rx Message Buffer receiving process successfully.
<i>kStatus_FLEXCAN_Rx-Busy</i>	- Rx Message Buffer is in use.

17.2.7.30 status_t FLEXCAN_TransferReceiveFifoNonBlocking (CAN_Type * *base*, flexcan_handle_t * *handle*, flexcan_fifo_transfer_t * *xfer*)

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>xfer</i>	FlexCAN Rx FIFO transfer structure. See the flexcan_fifo_transfer_t .

Return values

<i>kStatus_Success</i>	- Start Rx FIFO receiving process successfully.
<i>kStatus_FLEXCAN_Rx-FifoBusy</i>	- Rx FIFO is currently in use.

17.2.7.31 void FLEXCAN_TransferAbortSend (CAN_Type * *base*, flexcan_handle_t * *handle*, uint8_t *mbIdx*)

This function aborts the interrupt driven message send process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.

17.2.7.32 void FLEXCAN_TransferAbortReceive (CAN_Type * *base*, flexcan_handle_t * *handle*, uint8_t *mbIdx*)

This function aborts the interrupt driven message receive process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.

17.2.7.33 void FLEXCAN_TransferAbortReceiveFifo (CAN_Type * *base*, flexcan_handle_t * *handle*)

This function aborts the interrupt driven message receive from Rx FIFO process.

FlexCAN Driver

Parameters

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

17.2.7.34 void FLEXCAN_TransferHandleIRQ (CAN_Type * *base*, flexcan_handle_t * *handle*)

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

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

17.3 FlexCAN eDMA Driver

17.3.1 Overview

Data Structures

- struct `flexcan_edma_handle_t`
FlexCAN eDMA handle. [More...](#)

TypeDefs

- typedef void(* `flexcan_edma_transfer_callback_t`)
(CAN_Type *base, flexcan_edma_handle_t
*handle, status_t status, void *userData)
FlexCAN transfer callback function.

eDMA transactional

- void `FLEXCAN_TransferCreateHandleEDMA` (CAN_Type *base, flexcan_edma_handle_t
*handle, `flexcan_edma_transfer_callback_t` callback, void *userData, `edma_handle_t` *rxFifoEdmaHandle)
Initializes the FlexCAN handle, which is used in transactional functions.
- status_t `FLEXCAN_TransferReceiveFifoEDMA` (CAN_Type *base, flexcan_edma_handle_t
*handle, `flexcan_fifo_transfer_t` *xfer)
Receives the CAN Message from the Rx FIFO using eDMA.
- void `FLEXCAN_TransferAbortReceiveFifoEDMA` (CAN_Type *base, flexcan_edma_handle_t
*handle)
Aborts the receive process which used eDMA.

17.3.2 Data Structure Documentation

17.3.2.1 struct _flexcan_edma_handle

Data Fields

- `flexcan_edma_transfer_callback_t` `callback`
Callback function.
- void * `userData`
FlexCAN callback function parameter.
- `edma_handle_t` * `rxFifoEdmaHandle`
The EDMA Rx FIFO channel used.
- volatile uint8_t `rxFifoState`
Rx FIFO transfer state.

FlexCAN eDMA Driver

17.3.2.1.0.54 Field Documentation

17.3.2.1.0.54.1 `flexcan_edma_transfer_callback_t flexcan_edma_handle_t::callback`

17.3.2.1.0.54.2 `void* flexcan_edma_handle_t::userData`

17.3.2.1.0.54.3 `edma_handle_t* flexcan_edma_handle_t::rxFifoEdmaHandle`

17.3.2.1.0.54.4 `volatile uint8_t flexcan_edma_handle_t::rxFifoState`

17.3.3 Typedef Documentation

17.3.3.1 `typedef void(* flexcan_edma_transfer_callback_t)(CAN_Type *base,
flexcan_edma_handle_t *handle, status_t status, void *userData)`

17.3.4 Function Documentation

17.3.4.1 `void FLEXCAN_TransferCreateHandleEDMA (CAN_Type * base,
flexcan_edma_handle_t * handle, flexcan_edma_transfer_callback_t callback,
void * userData, edma_handle_t * rxFifoEdmaHandle)`

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	Pointer to flexcan_edma_handle_t structure.
<i>callback</i>	The callback function.
<i>userData</i>	The parameter of the callback function.
<i>rxFifoEdmaHandle</i>	User-requested DMA handle for Rx FIFO DMA transfer.

17.3.4.2 status_t FLEXCAN_TransferReceiveFifoEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*, flexcan_fifo_transfer_t * *xfer*)

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	Pointer to flexcan_edma_handle_t structure.
<i>xfer</i>	FlexCAN Rx FIFO EDMA transfer structure, see flexcan_fifo_transfer_t .

Return values

<i>kStatus_Success</i>	if succeed, others failed.
<i>kStatus_FLEXCAN_Rx-FifoBusy</i>	Previous transfer ongoing.

17.3.4.3 void FLEXCAN_TransferAbortReceiveFifoEDMA (CAN_Type * *base*, flexcan_edma_handle_t * *handle*)

This function aborts the receive process which used eDMA.

Parameters

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

FlexCAN eDMA Driver

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

Chapter 18

FTM: FlexTimer Driver

18.1 Overview

The KSDK provides a driver for the FlexTimer Module (FTM) of Kinetis devices.

18.2 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.

18.2.1 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.

18.2.2 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.

18.2.3 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.

18.2.4 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.

18.2.5 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.

18.2.6 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.

18.3 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)
```

18.4 Typical use case

18.4.1 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_S-
 HIFT)),
 kFTM_ClearOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (2U << FTM_CnSC_ELSA_SH-
 IFT)),
 kFTM_SetOnMatch = ((1U << FTM_CnSC_MSA_SHIFT) | (3U << FTM_CnSC_ELSA_SHIF-

- 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_QquadDecoderCountingIncreaseFlag = FTM_QDCTRL_QUADIR_MASK,
kFTM_QquadDecoderCountingOverflowOnTopFlag = 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.

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.

18.5 Data Structure Documentation

18.5.1 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.

Data Structure Documentation

- `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.

18.5.1.0.0.55 Field Documentation

18.5.1.0.0.55.1 `ftm_chnl_t ftm_chnl_pwm_signal_param_t::chnlNumber`

In combined mode, this represents the channel pair number.

18.5.1.0.0.55.2 `ftm_pwm_level_select_t ftm_chnl_pwm_signal_param_t::level`

18.5.1.0.0.55.3 `uint8_t ftm_chnl_pwm_signal_param_t::dutyCyclePercent`

100 = always active signal (100% duty cycle).

18.5.1.0.0.55.4 `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

18.5.2 `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.

18.5.3 `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.

18.5.4 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.

18.5.5 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.
- `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.

Enumeration Type Documentation

18.5.5.0.0.56 Field Documentation

18.5.5.0.0.56.1 `uint32_t ftm_config_t::pwmSyncMode`

18.5.5.0.0.56.2 `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](#).

18.5.5.0.0.56.3 `uint32_t ftm_config_t::deadTimeValue`

18.5.5.0.0.56.4 `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](#).

18.6 Enumeration Type Documentation

18.6.1 `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.

18.6.2 `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.

18.6.3 enum ftm_pwm_mode_t

Enumerator

- kFTM_EdgeAlignedPwm* Edge-aligned PWM.
- kFTM_CenterAlignedPwm* Center-aligned PWM.
- kFTM_CombinedPwm* Combined PWM.

18.6.4 enum ftm_pwm_level_select_t

Enumerator

- kFTM_NoPwmSignal* No PWM output on pin.
- kFTM_LowTrue* Low true pulses.
- kFTM_HighTrue* High true pulses.

18.6.5 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.

18.6.6 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.

18.6.7 enum ftm_dual_edge_capture_mode_t

Enumerator

- kFTM_OneShot* One-shot capture mode.
- kFTM_Continuous* Continuous capture mode.

Enumeration Type Documentation

18.6.8 enum ftm_quad_decode_mode_t

Enumerator

kFTM_QuadPhaseEncode Phase A and Phase B encoding mode.

kFTM_QuadCountAndDir Count and direction encoding mode.

18.6.9 enum ftm_phase_polarity_t

Enumerator

kFTM_QuadPhaseNormal Phase input signal is not inverted.

kFTM_QuadPhaseInvert Phase input signal is inverted.

18.6.10 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.

18.6.11 enum ftm_clock_source_t

Enumerator

kFTM_SystemClock System clock selected.

kFTM_FixedClock Fixed frequency clock.

kFTM_ExternalClock External clock.

18.6.12 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.

18.6.13 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.

18.6.14 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.

18.6.15 enum ftm_external_trigger_t

Note

Actual available external trigger sources are SoC-specific

Enumerator

kFTM_Chnl0Trigger Generate trigger when counter equals chnl 0 CnV reg.

kFTM_Chnl1Trigger Generate trigger when counter equals chnl 1 CnV reg.

kFTM_Chnl2Trigger Generate trigger when counter equals chnl 2 CnV reg.

kFTM_Chnl3Trigger Generate trigger when counter equals chnl 3 CnV reg.

kFTM_Chnl4Trigger Generate trigger when counter equals chnl 4 CnV reg.

kFTM_Chnl5Trigger Generate trigger when counter equals chnl 5 CnV reg.

kFTM_Chnl6Trigger Available on certain SoC's, generate trigger when counter equals chnl 6 CnV reg.

kFTM_Chnl7Trigger Available on certain SoC's, generate trigger when counter equals chnl 7 CnV reg.

kFTM_InitTrigger Generate Trigger when counter is updated with CNTIN.

kFTM_ReloadInitTrigger Available on certain SoC's, trigger on reload point.

Enumeration Type Documentation

18.6.16 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.

18.6.17 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.

18.6.18 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.

18.6.19 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.
kFTM_TimeOverflowFlag Time overflow Flag.
kFTM_ChnlTriggerFlag Channel trigger Flag.
kFTM_ReloadFlag Reload Flag; Available only on certain SoC's.

18.6.20 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.

18.7 Function Documentation

18.7.1 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.

Function Documentation

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.

18.7.2 void FTM_Deinit (FTM_Type * *base*)

Parameters

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

18.7.3 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.
---------------	--

18.7.4 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

18.7.5 void FTM_UpdatePwmDutycycle (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent*)

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel/channel pair number. In combined mode, this represents the channel pair number
<i>currentPwmMode</i>	The current PWM mode set during PWM setup
<i>dutyCyclePercent</i>	New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

18.7.6 void FTM_UpdateChnlEdgeLevelSelect (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, uint8_t *level*)

Parameters

Function Documentation

<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.

18.7.7 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.

18.7.8 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.

18.7.9 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.

18.7.10 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.

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

18.7.11 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

18.7.12 void FTM_DisableInterrupts (FTM_Type * *base*, uint32_t *mask*)

Function Documentation

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

18.7.13 `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](#)

18.7.14 `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](#)

18.7.15 `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

18.7.16 `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.

18.7.17 static void FTM_StopTimer (FTM_Type * *base*) [inline], [static]

Parameters

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

18.7.18 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

18.7.19 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

18.7.20 static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type * *base*, *bool enable*) [inline], [static]

Function Documentation

Parameters

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

18.7.21 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

18.7.22 static void FTM_SetFaultControlEnable (FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value*) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: Enable fault control for this channel pair; false: No fault control

18.7.23 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

18.7.24 **static void FTM_SetComplementaryEnable (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 complementary mode; false: disable complementary mode

18.7.25 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

18.7.26 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

18.7.27 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](#).

18.7.28 static void FTM_SetQuadDecoderModuloValue (FTM_Type * *base*, uint32_t *startValue*, uint32_t *overValue*) [inline], [static]

The modulo values configure the minimum and maximum values that the Quad decoder counter can reach. After the counter goes over, the counter value goes to the other side and decrease/increase again.

Parameters

<i>base</i>	FTM peripheral base address.
<i>startValue</i>	The low limit value for Quad Decoder counter.
<i>overValue</i>	The high limit value for Quad Decoder counter.

18.7.29 static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type * *base*) [inline], [static]

Parameters

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

Returns

Current quad Decoder counter value.

18.7.30 static void FTM_ClearQuadDecoderCounterValue (FTM_Type * *base*) [inline], [static]

The counter is set as the initial value.

Function Documentation

Parameters

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

**18.7.31 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

**18.7.32 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 19

GPIO: General-Purpose Input/Output Driver

19.1 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.

19.2 Data Structure Documentation

19.2.1 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

19.3 Macro Definition Documentation

19.3.1 `#define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))`

19.4 Enumeration Type Documentation

19.4.1 `enum gpio_pin_direction_t`

Enumerator

kGPIO_DigitalInput Set current pin as digital input.

kGPIO_DigitalOutput Set current pin as digital output.

19.5 GPIO Driver

19.5.1 Overview

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

19.5.2 Typical use case

19.5.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};

/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

19.5.2.2 Input Operation

```
/* Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_GPIO_PIN,
                           kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};

/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

GPIO Configuration

- void **GPIO_PinInit** (GPIO_Type *base, uint32_t pin, const **gpio_pin_config_t** *config)
Initializes a GPIO pin used by the board.

GPIO Output Operations

- static void **GPIO_WritePinOutput** (GPIO_Type *base, uint32_t pin, uint8_t output)
Sets the output level of the multiple GPIO pins to the logic 1 or 0.
- static void **GPIO_SetPinsOutput** (GPIO_Type *base, uint32_t mask)
Sets the output level of the multiple GPIO pins to the logic 1.
- static void **GPIO_ClearPinsOutput** (GPIO_Type *base, uint32_t mask)
Sets the output level of the multiple GPIO pins to the logic 0.
- static void **GPIO_TogglePinsOutput** (GPIO_Type *base, uint32_t mask)
Reverses the current output logic of the multiple GPIO pins.

GPIO Input Operations

- static uint32_t [GPIO_ReadPinInput](#) (GPIO_Type *base, uint32_t pin)
Reads the current input value of the GPIO port.

GPIO Interrupt

- uint32_t [GPIO_GetPinsInterruptFlags](#) (GPIO_Type *base)
Reads the GPIO port interrupt status flag.
- void [GPIO_ClearPinsInterruptFlags](#) (GPIO_Type *base, uint32_t mask)
Clears multiple GPIO pin interrupt status flags.

19.5.3 Function Documentation

19.5.3.1 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

19.5.3.2 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.

19.5.3.3 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

19.5.3.4 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

19.5.3.5 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

19.5.3.6 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.
-------------	---

19.5.3.7 `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.
------------	---

19.5.3.8 `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

19.6 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.

19.6.1 Typical use case

19.6.1.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};

/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

19.6.1.2 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 */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```


Chapter 20

I2C: Inter-Integrated Circuit Driver

20.1 Overview

Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver
- I2C µCOS/II Driver
- I2C µCOS/III Driver

I2C Driver

20.2 I2C Driver

20.2.1 Overview

The KSDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of Kinetis devices.

The I2C driver includes functional APIs and transactional APIs.

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

Transactional APIs target the high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code using the functional APIs or accessing the hardware registers.

Transactional APIs support asynchronous transfer. This means that the functions [I2C_MasterTransferNonBlocking\(\)](#) set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

20.2.2 Typical use case

20.2.2.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* Gets the default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

/* Sends a start and a slave address. */
I2C_MasterStart(EXAMPLE_I2C_MASTER_BASEADDR, 7-bit slave address,
    kI2C_Write/kI2C_Read);

/* Waits for the sent out address. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR)) & kI2C_IntPendingFlag))
{
}

if(status & kI2C_ReceiveNakFlag)
{
    return kStatus_I2C_Nak;
}

result = I2C_MasterWriteBlocking(EXAMPLE_I2C_MASTER_BASEADDR, txBuff, BUFFER_SIZE);

if(result)
{
    /* If an error occurs, send STOP. */
```

```

    I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
    return result;
}

while (!(I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR) & kI2C_IntPendingFlag))
{
}

/* Wait for all data to be sent out and sends STOP. */
I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);

```

20.2.2.2 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;

```

20.2.2.3 Master Operation in DMA transactional method

```

i2c_master_dma_handle_t g_m_dma_handle;
dma_handle_t dmaHandle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;

```

I2C Driver

```
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Gets the default configuration for the master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

DMAMGR_RequestChannel((dma_request_source_t)DMA_REQUEST_SRC, 0, &dmaHandle);

I2C_MasterTransferCreateHandleDMA(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_dma_handle, i2c_master_callback, NULL, &dmaHandle);
I2C_MasterTransferDMA(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_dma_handle, &masterXfer);

/* Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

20.2.2.4 Slave Operation in functional method

```
i2c_slave_config_t slaveConfig;
uint8_t status;
status_t result = kStatus_Success;

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
                                             addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
    kI2C_RangeMatch;
I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

/* Waits for an address match. */
while (!(status = I2C_GetStatusFlag(EXAMPLE_I2C_SLAVE_BASEADDR)) & kI2C_AddressMatchFlag)
{

}

/* A slave transmits; master is reading from the slave. */
if (status & kI2C_TransferDirectionFlag)
{
    result = I2C_SlaveWriteBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}
else
{
```

```

    I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}

return result;
}

20.2.2.5 Slave Operation in interrupt transactional method

i2c_slave_config_t slaveConfig;
i2c_slave_handle_t g_s_handle;
volatile bool g_SlaveCompletionFlag = false;

static void i2c_slave_callback(I2C_Type *base, i2c_slave_transfer_t *xfer, void *
    userData)
{
    switch (xfer->event)
    {
        /* Transmit request */
        case kI2C_SlaveTransmitEvent:
            /* Update information for transmit process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Receives request */
        case kI2C_SlaveReceiveEvent:
            /* Update information for received process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Transfer is done */
        case kI2C_SlaveCompletionEvent:
            g_SlaveCompletionFlag = true;
            break;

        default:
            g_SlaveCompletionFlag = true;
            break;
    }
}

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
    addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
    kI2C_RangeMatch;

I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

I2C_SlaveTransferCreateHandle(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
    i2c_slave_callback, NULL);

I2C_SlaveTransferNonBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
    kI2C_SlaveCompletionEvent);

/* Waits for a transfer to be completed. */
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;

```

I2C Driver

Data Structures

- struct `i2c_master_config_t`
I2C master user configuration. [More...](#)
- struct `i2c_slave_config_t`
I2C slave user configuration. [More...](#)
- struct `i2c_master_transfer_t`
I2C master transfer structure. [More...](#)
- struct `i2c_master_handle_t`
I2C master handle structure. [More...](#)
- struct `i2c_slave_transfer_t`
I2C slave transfer structure. [More...](#)
- struct `i2c_slave_handle_t`
I2C slave handle structure. [More...](#)

Typedefs

- typedef void(* `i2c_master_transfer_callback_t`)
(I2C_Type *base, i2c_master_handle_t *handle,
status_t status, void *userData)
I2C master transfer callback typedef.
- typedef void(* `i2c_slave_transfer_callback_t`)
(I2C_Type *base, i2c_slave_transfer_t *xfer, void
*userData)
I2C slave transfer callback typedef.

Enumerations

- enum `_i2c_status` {
kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_I2C, 0),
kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_I2C, 1),
kStatus_I2C_Nak = MAKE_STATUS(kStatusGroup_I2C, 2),
kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_I2C, 3),
kStatus_I2C_Timeout = MAKE_STATUS(kStatusGroup_I2C, 4),
kStatus_I2C_Addr_Nak = MAKE_STATUS(kStatusGroup_I2C, 5) }
I2C status return codes.
- enum `_i2c_flags` {
kI2C_ReceiveNakFlag = I2C_S_RXAK_MASK,
kI2C_IntPendingFlag = I2C_S_IICIF_MASK,
kI2C_TransferDirectionFlag = I2C_S_SRW_MASK,
kI2C_RangeAddressMatchFlag = I2C_S_RAM_MASK,
kI2C_ArbitrationLostFlag = I2C_S_ARBL_MASK,
kI2C_BusBusyFlag = I2C_S_BUSY_MASK,
kI2C_AddressMatchFlag = I2C_S_IAAS_MASK,
kI2C_TransferCompleteFlag = I2C_S_TCF_MASK,
kI2C_StopDetectFlag = I2C_FLT_STOPF_MASK << 8,
kI2C_StartDetectFlag = I2C_FLT_STARTF_MASK << 8 }
I2C peripheral flags.

- enum `_i2c_interrupt_enable` {

 `kI2C_GlobalInterruptEnable` = I2C_C1_IICIE_MASK,

 `kI2C_StartStopDetectInterruptEnable` = I2C_FLT_SSIE_MASK }

I2C feature interrupt source.
- enum `i2c_direction_t` {

 `kI2C_Write` = 0x0U,

 `kI2C_Read` = 0x1U }

The direction of master and slave transfers.
- enum `i2c_slave_address_mode_t` {

 `kI2C_Address7bit` = 0x0U,

 `kI2C_RangeMatch` = 0X2U }

Addressing mode.
- enum `_i2c_master_transfer_flags` {

 `kI2C_TransferDefaultFlag` = 0x0U,

 `kI2C_TransferNoStartFlag` = 0x1U,

 `kI2C_TransferRepeatedStartFlag` = 0x2U,

 `kI2C_TransferNoStopFlag` = 0x4U }

I2C transfer control flag.
- enum `i2c_slave_transfer_event_t` {

 `kI2C_SlaveAddressMatchEvent` = 0x01U,

 `kI2C_SlaveTransmitEvent` = 0x02U,

 `kI2C_SlaveReceiveEvent` = 0x04U,

 `kI2C_SlaveTransmitAckEvent` = 0x08U,

 `kI2C_SlaveStartEvent` = 0x10U,

 `kI2C_SlaveCompletionEvent` = 0x20U,

 `kI2C_SlaveGenaralcallEvent` = 0x40U,

 `kI2C_SlaveAllEvents` }

Set of events sent to the callback for nonblocking slave transfers.

Driver version

- #define `FSL_I2C_DRIVER_VERSION` (MAKE_VERSION(2, 0, 2))

I2C driver version 2.0.2.

Initialization and deinitialization

- void `I2C_MasterInit` (I2C_Type *base, const `i2c_master_config_t` *masterConfig, uint32_t srcClock_Hz)

Initializes the I2C peripheral.
- void `I2C_SlaveInit` (I2C_Type *base, const `i2c_slave_config_t` *slaveConfig, uint32_t srcClock_Hz)

Initializes the I2C peripheral.
- void `I2C_MasterDeinit` (I2C_Type *base)

De-initializes the I2C master peripheral.
- void `I2C_SlaveDeinit` (I2C_Type *base)

I2C Driver

- **void I2C_MasterGetDefaultConfig (i2c_master_config_t *masterConfig)**
De-initializes the I2C slave peripheral.
Sets the I2C master configuration structure to default values.
- **void I2C_SlaveGetDefaultConfig (i2c_slave_config_t *slaveConfig)**
Sets the I2C slave configuration structure to default values.
- **static void I2C_Enable (I2C_Type *base, bool enable)**
Enables or disables the I2C peripheral operation.

Status

- **uint32_t I2C_MasterGetStatusFlags (I2C_Type *base)**
Gets the I2C status flags.
- **static uint32_t I2C_SlaveGetStatusFlags (I2C_Type *base)**
Gets the I2C status flags.
- **static void I2C_MasterClearStatusFlags (I2C_Type *base, uint32_t statusMask)**
Clears the I2C status flag state.
- **static void I2C_SlaveClearStatusFlags (I2C_Type *base, uint32_t statusMask)**
Clears the I2C status flag state.

Interrupts

- **void I2C_EnableInterrupts (I2C_Type *base, uint32_t mask)**
Enables I2C interrupt requests.
- **void I2C_DisableInterrupts (I2C_Type *base, uint32_t mask)**
Disables I2C interrupt requests.

DMA Control

- **static void I2C_EnableDMA (I2C_Type *base, bool enable)**
Enables/disables the I2C DMA interrupt.
- **static uint32_t I2C_GetDataRegAddr (I2C_Type *base)**
Gets the I2C tx/rx data register address.

Bus Operations

- **void I2C_MasterSetBaudRate (I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)**
Sets the I2C master transfer baud rate.
- **status_t I2C_MasterStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)**
Sends a START on the I2C bus.
- **status_t I2C_MasterStop (I2C_Type *base)**
Sends a STOP signal on the I2C bus.
- **status_t I2C_MasterRepeatedStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)**
Sends a REPEATED START on the I2C bus.
- **status_t I2C_MasterWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize, uint32_t flags)**
Performs a polling send transaction on the I2C bus.

- status_t **I2C_MasterReadBlocking** (I2C_Type *base, uint8_t *rxBuff, size_t rxSize, uint32_t flags)
Performs a polling receive transaction on the I2C bus.
- status_t **I2C_SlaveWriteBlocking** (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
Performs a polling send transaction on the I2C bus.
- void **I2C_SlaveReadBlocking** (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
Performs a polling receive transaction on the I2C bus.
- status_t **I2C_MasterTransferBlocking** (I2C_Type *base, i2c_master_transfer_t *xfer)
Performs a master polling transfer on the I2C bus.

Transactional

- void **I2C_MasterTransferCreateHandle** (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_callback_t callback, void *userData)
Initializes the I2C handle which is used in transactional functions.
- status_t **I2C_MasterTransferNonBlocking** (I2C_Type *base, i2c_master_handle_t *handle, i2c_master_transfer_t *xfer)
Performs a master interrupt non-blocking transfer on the I2C bus.
- status_t **I2C_MasterTransferGetCount** (I2C_Type *base, i2c_master_handle_t *handle, size_t *count)
Gets the master transfer status during a interrupt non-blocking transfer.
- void **I2C_MasterTransferAbort** (I2C_Type *base, i2c_master_handle_t *handle)
Aborts an interrupt non-blocking transfer early.
- void **I2C_MasterTransferHandleIRQ** (I2C_Type *base, void *i2cHandle)
Master interrupt handler.
- void **I2C_SlaveTransferCreateHandle** (I2C_Type *base, i2c_slave_handle_t *handle, i2c_slave_transfer_callback_t callback, void *userData)
Initializes the I2C handle which is used in transactional functions.
- status_t **I2C_SlaveTransferNonBlocking** (I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)
Starts accepting slave transfers.
- void **I2C_SlaveTransferAbort** (I2C_Type *base, i2c_slave_handle_t *handle)
Aborts the slave transfer.
- status_t **I2C_SlaveTransferGetCount** (I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)
Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.
- void **I2C_SlaveTransferHandleIRQ** (I2C_Type *base, void *i2cHandle)
Slave interrupt handler.

20.2.3 Data Structure Documentation

20.2.3.1 struct i2c_master_config_t

Data Fields

- bool **enableMaster**
Enables the I2C peripheral at initialization time.
- bool **enableHighDrive**
Controls the drive capability of the I2C pads.

I2C Driver

- bool `enableStopHold`
Controls the stop hold enable.
- uint32_t `baudRate_Bps`
Baud rate configuration of I2C peripheral.
- uint8_t `glitchFilterWidth`
Controls the width of the glitch.

20.2.3.1.0.57 Field Documentation

20.2.3.1.0.57.1 `bool i2c_master_config_t::enableMaster`

20.2.3.1.0.57.2 `bool i2c_master_config_t::enableHighDrive`

20.2.3.1.0.57.3 `bool i2c_master_config_t::enableStopHold`

20.2.3.1.0.57.4 `uint32_t i2c_master_config_t::baudRate_Bps`

20.2.3.1.0.57.5 `uint8_t i2c_master_config_t::glitchFilterWidth`

20.2.3.2 struct `i2c_slave_config_t`

Data Fields

- bool `enableSlave`
Enables the I2C peripheral at initialization time.
- bool `enableGeneralCall`
Enables the general call addressing mode.
- bool `enableWakeUp`
Enables/disables waking up MCU from low-power mode.
- bool `enableHighDrive`
Controls the drive capability of the I2C pads.
- bool `enableBaudRateCtl`
Enables/disables independent slave baud rate on SCL in very fast I2C modes.
- uint16_t `slaveAddress`
A slave address configuration.
- uint16_t `upperAddress`
A maximum boundary slave address used in a range matching mode.
- `i2c_slave_address_mode_t addressingMode`
An addressing mode configuration of `i2c_slave_address_mode_config_t`.
- uint32_t `sclStopHoldTime_ns`
the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.

20.2.3.2.0.58 Field Documentation

- 20.2.3.2.0.58.1 `bool i2c_slave_config_t::enableSlave`
- 20.2.3.2.0.58.2 `bool i2c_slave_config_t::enableGeneralCall`
- 20.2.3.2.0.58.3 `bool i2c_slave_config_t::enableWakeUp`
- 20.2.3.2.0.58.4 `bool i2c_slave_config_t::enableHighDrive`
- 20.2.3.2.0.58.5 `bool i2c_slave_config_t::enableBaudRateCtl`
- 20.2.3.2.0.58.6 `uint16_t i2c_slave_config_t::slaveAddress`
- 20.2.3.2.0.58.7 `uint16_t i2c_slave_config_t::upperAddress`
- 20.2.3.2.0.58.8 `i2c_slave_address_mode_t i2c_slave_config_t::addressingMode`
- 20.2.3.2.0.58.9 `uint32_t i2c_slave_config_t::sclStopHoldTime_ns`

20.2.3.3 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.

20.2.3.3.0.59 Field Documentation

- 20.2.3.3.0.59.1 `uint32_t i2c_master_transfer_t::flags`
- 20.2.3.3.0.59.2 `uint8_t i2c_master_transfer_t::slaveAddress`
- 20.2.3.3.0.59.3 `i2c_direction_t i2c_master_transfer_t::direction`
- 20.2.3.3.0.59.4 `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

I2C Driver

20.2.3.3.0.59.5 uint8_t i2c_master_transfer_t::subaddressSize

20.2.3.3.0.59.6 uint8_t* volatile i2c_master_transfer_t::data

20.2.3.3.0.59.7 volatile size_t i2c_master_transfer_t::dataSize

20.2.3.4 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.

20.2.3.4.0.60 Field Documentation

20.2.3.4.0.60.1 i2c_master_transfer_t i2c_master_handle_t::transfer

20.2.3.4.0.60.2 size_t i2c_master_handle_t::transferSize

20.2.3.4.0.60.3 uint8_t i2c_master_handle_t::state

20.2.3.4.0.60.4 i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback

20.2.3.4.0.60.5 void* i2c_master_handle_t::userData

20.2.3.5 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.

20.2.3.5.0.61 Field Documentation

20.2.3.5.0.61.1 `i2c_slave_transfer_event_t i2c_slave_transfer_t::event`

20.2.3.5.0.61.2 `uint8_t* volatile i2c_slave_transfer_t::data`

20.2.3.5.0.61.3 `volatile size_t i2c_slave_transfer_t::dataSize`

20.2.3.5.0.61.4 `status_t i2c_slave_transfer_t::completionStatus`

Only applies for [kI2C_SlaveCompletionEvent](#).

20.2.3.5.0.61.5 `size_t i2c_slave_transfer_t::transferredCount`

20.2.3.6 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

20.2.3.6.0.62 Field Documentation

20.2.3.6.0.62.1 **volatile bool i2c_slave_handle_t::isBusy**

20.2.3.6.0.62.2 **i2c_slave_transfer_t i2c_slave_handle_t::transfer**

20.2.3.6.0.62.3 **uint32_t i2c_slave_handle_t::eventMask**

20.2.3.6.0.62.4 **i2c_slave_transfer_callback_t i2c_slave_handle_t::callback**

20.2.3.6.0.62.5 **void* i2c_slave_handle_t::userData**

20.2.4 Macro Definition Documentation

20.2.4.1 **#define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))**

20.2.5 Typedef Documentation

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

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

20.2.6 Enumeration Type Documentation

20.2.6.1 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.

20.2.6.2 enum _i2c_flags

The following status register flags can be cleared:

- [kI2C_ArbitrationLostFlag](#)
- [kI2C_IntPendingFlag](#)
- [kI2C_StartDetectFlag](#)
- [kI2C_StopDetectFlag](#)

Note

These enumerations are meant to be OR'd together to form a bit mask.

Enumerator

- kI2C_ReceiveNakFlag* I2C receive NAK flag.
- kI2C_IntPendingFlag* I2C interrupt pending flag.
- kI2C_TransferDirectionFlag* I2C transfer direction flag.
- kI2C_RangeAddressMatchFlag* I2C range address match flag.
- kI2C_ArbitrationLostFlag* I2C arbitration lost flag.
- kI2C_BusBusyFlag* I2C bus busy flag.
- kI2C_AddressMatchFlag* I2C address match flag.
- kI2C_TransferCompleteFlag* I2C transfer complete flag.
- kI2C_StopDetectFlag* I2C stop detect flag.
- kI2C_StartDetectFlag* I2C start detect flag.

20.2.6.3 enum _i2c_interrupt_enable

Enumerator

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

20.2.6.4 enum i2c_direction_t

Enumerator

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

20.2.6.5 enum i2c_slave_address_mode_t

Enumerator

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

20.2.6.6 enum _i2c_master_transfer_flags

Enumerator

- kI2C_TransferDefaultFlag* A transfer starts with a start signal, stops with a stop signal.

I2C Driver

kI2C_TransferNoStartFlag A transfer starts without a start signal.

kI2C_TransferRepeatedStartFlag A transfer starts with a repeated start signal.

kI2C_TransferNoStopFlag A transfer ends without a stop signal.

20.2.6.7 enum i2c_slave_transfer_event_t

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to [I2C_SlaveTransferNonBlocking\(\)](#) to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note

These enumerations are meant to be OR'd together to form a bit mask of events.

Enumerator

kI2C_SlaveAddressMatchEvent Received the slave address after a start or repeated start.

kI2C_SlaveTransmitEvent A callback is requested to provide data to transmit (slave-transmitter role).

kI2C_SlaveReceiveEvent A callback is requested to provide a buffer in which to place received data (slave-receiver role).

kI2C_SlaveTransmitAckEvent A callback needs to either transmit an ACK or NACK.

kI2C_SlaveStartEvent A start/repeated start was detected.

kI2C_SlaveCompletionEvent A stop was detected or finished transfer, completing the transfer.

kI2C_SlaveGeneralCallEvent Received the general call address after a start or repeated start.

kI2C_SlaveAllEvents A bit mask of all available events.

20.2.7 Function Documentation

20.2.7.1 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

20.2.7.2 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

20.2.7.3 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
-------------	------------------

20.2.7.4 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
-------------	------------------

20.2.7.5 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.
---------------------	--

20.2.7.6 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.
--------------------	---

20.2.7.7 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.

20.2.7.8 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.

20.2.7.9 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.

20.2.7.10 static void I2C_MasterClearStatusFlags (I2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag.

I2C Driver

Parameters

<i>base</i>	I2C base pointer
<i>statusMask</i>	The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values: <ul style="list-style-type: none">• kI2C_StartDetectFlag (if available)• kI2C_StopDetectFlag (if available)• kI2C_ArbitrationLostFlag• kI2C_IntPendingFlagFlag

20.2.7.11 static void I2C_SlaveClearStatusFlags (I2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag

Parameters

<i>base</i>	I2C base pointer
<i>statusMask</i>	The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values: <ul style="list-style-type: none">• kI2C_StartDetectFlag (if available)• kI2C_StopDetectFlag (if available)• kI2C_ArbitrationLostFlag• kI2C_IntPendingFlagFlag

20.2.7.12 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">• kI2C_GlobalInterruptEnable• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable• kI2C_SdaTimeoutInterruptEnable

20.2.7.13 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

20.2.7.14 static void I2C_EnableDMA (I2C_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	I2C base pointer
<i>enable</i>	true to enable, false to disable

20.2.7.15 static uint32_t I2C_GetDataRegAddr (I2C_Type * *base*) [inline], [static]

This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

<i>base</i>	I2C base pointer
-------------	------------------

Returns

data register address

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

Parameters

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

I2C Driver

20.2.7.17 **status_t I2C_MasterStart(I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*)**

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

<i>base</i>	I2C peripheral base pointer
<i>address</i>	7-bit slave device address.
<i>direction</i>	Master transfer directions(transmit/receive).

Return values

<i>kStatus_Success</i>	Successfully send the start signal.
<i>kStatus_I2C_Busy</i>	Current bus is busy.

20.2.7.18 **status_t I2C_MasterStop (I2C_Type * *base*)**

Return values

<i>kStatus_Success</i>	Successfully send the stop signal.
<i>kStatus_I2C_Timeout</i>	Send stop signal failed, timeout.

20.2.7.19 **status_t I2C_MasterRepeatedStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*)**

Parameters

<i>base</i>	I2C peripheral base pointer
<i>address</i>	7-bit slave device address.
<i>direction</i>	Master transfer directions(transmit/receive).

Return values

<i>kStatus_Success</i>	Successfully send the start signal.
<i>kStatus_I2C_Busy</i>	Current bus is busy but not occupied by current I2C master.

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

I2C Driver

Parameters

<i>base</i>	The I2C peripheral base pointer.
<i>txBuff</i>	The pointer to the data to be transferred.
<i>txSize</i>	The length in bytes of the data to be transferred.
<i>flags</i>	Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_ArbitrationLost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

20.2.7.21 **status_t I2C_MasterReadBlocking (I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*, uint32_t *flags*)**

Note

The I2C_MasterReadBlocking function stops the bus before reading the final byte. Without stopping the bus prior for the final read, the bus issues another read, resulting in garbage data being read into the data register.

Parameters

<i>base</i>	I2C peripheral base pointer.
<i>rxBuff</i>	The pointer to the data to store the received data.
<i>rxSize</i>	The length in bytes of the data to be received.
<i>flags</i>	Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_Timeout</i>	Send stop signal failed, timeout.

20.2.7.22 **status_t I2C_SlaveWriteBlocking (I2C_Type * *base*, const uint8_t * *txBuff*, size_t *txSize*)**

Parameters

<i>base</i>	The I2C peripheral base pointer.
<i>txBuff</i>	The pointer to the data to be transferred.
<i>txSize</i>	The length in bytes of the data to be transferred.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_ArbitrationLost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

20.2.7.23 void I2C_SlaveReadBlocking (I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*)

Parameters

<i>base</i>	I2C peripheral base pointer.
<i>rxBuff</i>	The pointer to the data to store the received data.
<i>rxSize</i>	The length in bytes of the data to be received.

20.2.7.24 status_t I2C_MasterTransferBlocking (I2C_Type * *base*, i2c_master_transfer_t * *xfer*)

Note

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

Parameters

<i>base</i>	I2C peripheral base address.
<i>xfer</i>	Pointer to the transfer structure.

Return values

I2C Driver

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_Busy</i>	Previous transmission still not finished.
<i>kStatus_I2C_Timeout</i>	Transfer error, wait signal timeout.
<i>kStatus_I2C_ArbitrationLost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

20.2.7.25 void I2C_MasterTransferCreateHandle (*I2C_Type * base*, *i2c_master_handle_t * handle*, *i2c_master_transfer_callback_t callback*, *void * userData*)

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to <i>i2c_master_handle_t</i> structure to store the transfer state.
<i>callback</i>	pointer to user callback function.
<i>userData</i>	user parameter passed to the callback function.

20.2.7.26 status_t I2C_MasterTransferNonBlocking (*I2C_Type * base*, *i2c_master_handle_t * handle*, *i2c_master_transfer_t * xfer*)

Note

Calling the API returns immediately after transfer initiates. The user needs to call *I2C_MasterGetTransferCount* to poll the transfer status to check whether the transfer is finished. If the return status is not *kStatus_I2C_Busy*, the transfer is finished.

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to <i>i2c_master_handle_t</i> structure which stores the transfer state.
<i>xfer</i>	pointer to <i>i2c_master_transfer_t</i> structure.

Return values

<i>kStatus_Success</i>	Successfully start the data transmission.
<i>kStatus_I2C_Busy</i>	Previous transmission still not finished.
<i>kStatus_I2C_Timeout</i>	Transfer error, wait signal timeout.

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

I2C Driver

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_master_handle_t structure which stores the transfer state.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

Return values

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

20.2.7.28 void I2C_MasterTransferAbort (I2C_Type * *base*, i2c_master_handle_t * *handle*)

Note

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

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_master_handle_t structure which stores the transfer state

20.2.7.29 void I2C_MasterTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Parameters

<i>base</i>	I2C base pointer.
<i>i2cHandle</i>	pointer to i2c_master_handle_t structure.

20.2.7.30 void I2C_SlaveTransferCreateHandle (I2C_Type * *base*, i2c_slave_handle_t * *handle*, i2c_slave_transfer_callback_t *callback*, void * *userData*)

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_slave_handle_t structure to store the transfer state.
<i>callback</i>	pointer to user callback function.
<i>userData</i>	user parameter passed to the callback function.

20.2.7.31 **status_t I2C_SlaveTransferNonBlocking (I2C_Type * *base*, i2c_slave_handle_t * *handle*, uint32_t *eventMask*)**

Call this API after calling the [I2C_SlaveInit\(\)](#) and [I2C_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to [I2C_SlaveTransferCreateHandle\(\)](#). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c_slave_transfer_event_t](#) enumerators for the events you wish to receive. The [k_I2C_SlaveTransmitEvent](#) and #[kLPI2C_SlaveReceiveEvent](#) events are always enabled and do not need to be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [kI2C_SlaveAllEvents](#) constant is provided as a convenient way to enable all events.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to # 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

#kStatus_Success	Slave transfers were successfully started.
kStatus_I2C_Busy	Slave transfers have already been started on this handle.

20.2.7.32 **void I2C_SlaveTransferAbort (I2C_Type * *base*, i2c_slave_handle_t * *handle*)**

Note

This API can be called at any time to stop slave for handling the bus events.

I2C Driver

Parameters

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

20.2.7.33 **status_t I2C_SlaveTransferGetCount (I2C_Type * *base*, i2c_slave_handle_t * *handle*, size_t * *count*)**

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_slave_handle_t structure.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

Return values

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

20.2.7.34 **void I2C_SlaveTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)**

Parameters

<i>base</i>	I2C base pointer.
<i>i2cHandle</i>	pointer to i2c_slave_handle_t structure which stores the transfer state

20.3 I2C eDMA Driver

20.3.1 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.

20.3.2 Data Structure Documentation

20.3.2.1 struct _i2c_master_edma_handle

I2C master eDMA handle typedef.

Data Fields

- [i2c_master_transfer_t transfer](#)
I2C master transfer structure.
- [size_t transferSize](#)
Total bytes to be transferred.
- [uint8_t nbytes](#)
eDMA minor byte transfer count initially configured.
- [uint8_t state](#)

I2C eDMA Driver

- I2C master transfer status.*
- **edma_handle_t * dmaHandle**
The eDMA handler used.
 - **i2c_master_edma_transfer_callback_t completionCallback**
A callback function called after the eDMA transfer is finished.
 - **void * userData**
A callback parameter passed to the callback function.

20.3.2.1.0.63 Field Documentation

20.3.2.1.0.63.1 i2c_master_transfer_t i2c_master_edma_handle_t::transfer

20.3.2.1.0.63.2 size_t i2c_master_edma_handle_t::transferSize

20.3.2.1.0.63.3 uint8_t i2c_master_edma_handle_t::nbytes

20.3.2.1.0.63.4 uint8_t i2c_master_edma_handle_t::state

20.3.2.1.0.63.5 edma_handle_t* i2c_master_edma_handle_t::dmaHandle

20.3.2.1.0.63.6 i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completionCallback

20.3.2.1.0.63.7 void* i2c_master_edma_handle_t::userData

20.3.3 Typedef Documentation

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

20.3.4 Function Documentation

20.3.4.1 `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 i2c_master_edma_handle_t structure.
<i>callback</i>	A pointer to the user callback function.

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

20.3.4.2 status_t I2C_MasterTransferEDMA (*I2C_Type* * *base*, *i2c_master_edma_handle_t* * *handle*, *i2c_master_transfer_t* * *xfer*)

Parameters

<i>base</i>	I2C peripheral base address.
<i>handle</i>	A pointer to the <i>i2c_master_edma_handle_t</i> structure.
<i>xfer</i>	A pointer to the transfer structure of <i>i2c_master_transfer_t</i> .

Return values

<i>kStatus_Success</i>	Sucessfully completed the data transmission.
<i>kStatus_I2C_Busy</i>	A previous transmission is still not finished.
<i>kStatus_I2C_Timeout</i>	Transfer error, waits for a signal timeout.
<i>kStatus_I2C_Arbitration_Lost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

20.3.4.3 status_t I2C_MasterTransferGetCountEDMA (*I2C_Type* * *base*, *i2c_master_edma_handle_t* * *handle*, *size_t* * *count*)

Parameters

<i>base</i>	I2C peripheral base address.
<i>handle</i>	A pointer to the <i>i2c_master_edma_handle_t</i> structure.
<i>count</i>	A number of bytes transferred by the non-blocking transaction.

20.3.4.4 void I2C_MasterTransferAbortEDMA (*I2C_Type* * *base*, *i2c_master_edma_handle_t* * *handle*)

I2C eDMA Driver

Parameters

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

20.4 I2C DMA Driver

20.4.1 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.

20.4.2 Data Structure Documentation

20.4.2.1 struct _i2c_master_dma_handle

I2C master DMA handle typedef.

Data Fields

- [i2c_master_transfer_t transfer](#)
I2C master transfer struct.
- [size_t transferSize](#)
Total bytes to be transferred.
- [uint8_t state](#)
I2C master transfer status.
- [dma_handle_t * dmaHandle](#)

I2C DMA Driver

The DMA handler used.

- **i2c_master_dma_transfer_callback_t completionCallback**
A callback function called after the DMA transfer finished.
- **void *userData**
A callback parameter passed to the callback function.

20.4.2.1.0.64 Field Documentation

20.4.2.1.0.64.1 i2c_master_transfer_t i2c_master_dma_handle_t::transfer

20.4.2.1.0.64.2 size_t i2c_master_dma_handle_t::transferSize

20.4.2.1.0.64.3 uint8_t i2c_master_dma_handle_t::state

20.4.2.1.0.64.4 dma_handle_t* i2c_master_dma_handle_t::dmaHandle

20.4.2.1.0.64.5 i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completionCallback

20.4.2.1.0.64.6 void* i2c_master_dma_handle_t::userData

20.4.3 Typedef Documentation

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

20.4.4 Function Documentation

20.4.4.1 void I2C_MasterTransferCreateHandleDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*, i2c_master_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	Pointer to the i2c_master_dma_handle_t structure
<i>callback</i>	Pointer to the user callback function
<i>userData</i>	A user parameter passed to the callback function
<i>dmaHandle</i>	DMA handle pointer

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

Parameters

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

Return values

<i>kStatus_Success</i>	Sucessfully completes the data transmission.
<i>kStatus_I2C_Busy</i>	A previous transmission is still not finished.
<i>kStatus_I2C_Timeout</i>	A transfer error, waits for the signal timeout.
<i>kStatus_I2C_Arbitration-Lost</i>	A transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	A transfer error, receives NAK during transfer.

20.4.4.3 status_t I2C_MasterTransferGetCountDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	A pointer to the i2c_master_dma_handle_t structure
<i>count</i>	A number of bytes transferred so far by the non-blocking transaction.

20.4.4.4 void I2C_MasterTransferAbortDMA (I2C_Type * *base*, i2c_master_dma_handle_t * *handle*)

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	A pointer to the i2c_master_dma_handle_t structure.

I2C FreeRTOS Driver

20.5 I2C FreeRTOS Driver

20.5.1 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.

20.5.2 Function Documentation

20.5.2.1 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.

20.5.2.2 status_t I2C_RTOS_Deinit (*i2c_rtos_handle_t * handle*)

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

Parameters

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

20.5.2.3 **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.

20.6 I2C μCOS/II Driver

20.6.1 Overview

I2C RTOS Operation

- status_t [I2C_RRTOS_Init](#) (i2c_rtos_handle_t *handle, I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
Initializes the I2C.
- status_t [I2C_RRTOS_Deinit](#) (i2c_rtos_handle_t *handle)
Deinitializes the I2C.
- status_t [I2C_RRTOS_Transfer](#) (i2c_rtos_handle_t *handle, i2c_master_transfer_t *transfer)
Performs the I2C transfer.

20.6.2 Function Documentation

20.6.2.1 status_t I2C_RRTOS_Init (*i2c_rtos_handle_t * handle, I2C_Type * base, const i2c_master_config_t * masterConfig, uint32_t srcClock_Hz*)

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

Parameters

<i>handle</i>	The RTOS I2C handle; the pointer to an allocated space for RTOS context.
<i>base</i>	The pointer base address of the I2C instance to initialize.
<i>masterConfig</i>	A configuration structure to set-up the I2C in master mode.
<i>srcClock_Hz</i>	A frequency of the input clock of the I2C module.

Returns

status of the operation.

20.6.2.2 status_t I2C_RRTOS_Deinit (*i2c_rtos_handle_t * handle*)

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

Parameters

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

20.6.2.3 **status_t I2C_RTOS_Transfer(i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)**

This function performs the I2C transfer according to the data given in the transfer structure.

Parameters

<i>handle</i>	The RTOS I2C handle.
<i>transfer</i>	A structure specifying the transfer parameters.

Returns

status of the operation.

20.7 I2C μCOS/III Driver

20.7.1 Overview

I2C RTOS Operation

- status_t [I2C_RTOS_Init](#) (i2c_rtos_handle_t *handle, I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
Initializes the I2C.
- status_t [I2C_RTOS_Deinit](#) (i2c_rtos_handle_t *handle)
Deinitializes the I2C.
- status_t [I2C_RTOS_Transfer](#) (i2c_rtos_handle_t *handle, i2c_master_transfer_t *transfer)
Performs the I2C transfer.

20.7.2 Function Documentation

20.7.2.1 status_t I2C_RTOS_Init (*i2c_rtos_handle_t * handle, I2C_Type * base, const i2c_master_config_t * masterConfig, uint32_t srcClock_Hz*)

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

Parameters

<i>handle</i>	The RTOS I2C handle; the pointer to an allocated space for the RTOS context.
<i>base</i>	The pointer base address of the I2C instance to initialize.
<i>masterConfig</i>	A configuration structure to set-up the I2C in master mode.
<i>srcClock_Hz</i>	A frequency of the input clock of the I2C module.

Returns

status of the operation.

20.7.2.2 status_t I2C_RTOS_Deinit (*i2c_rtos_handle_t * handle*)

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

Parameters

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

20.7.2.3 **status_t I2C_RTOS_Transfer(i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)**

This function performs the I2C transfer according to the data given in the transfer structure.

Parameters

<i>handle</i>	The RTOS I2C handle.
<i>transfer</i>	A structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 21

LLWU: Low-Leakage Wakeup Unit Driver

21.1 Overview

The KSDK provides a peripheral driver for the Low-Leakage Wakeup Unit (LLWU) module of Kinetis devices. The LLWU module allows the user to select external pin sources and internal modules as a wake-up source from low-leakage power modes.

21.2 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.

21.3 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.

21.4 Digital pin filter for external wakeup pin configurations

Configures the digital pin filter of the external wakeup pins' working modes, gets, and clears the pin filter flags. Digital pin filters are accessed by the `filterIndex`, which is started from 1. Numbers of external pins depend on the SoC configuration.

Data Structures

- struct `llwu_external_pin_filter_mode_t`
An external input pin filter control structure. [More...](#)

Enumerations

- enum `llwu_external_pin_mode_t` {
 `kLLWU_ExternalPinDisable` = 0U,
 `kLLWU_ExternalPinRisingEdge` = 1U,
 `kLLWU_ExternalPinFallingEdge` = 2U,
 `kLLWU_ExternalPinAnyEdge` = 3U }
External input pin control modes.
- enum `llwu_pin_filter_mode_t` {
 `kLLWU_PinFilterDisable` = 0U,
 `kLLWU_PinFilterRisingEdge` = 1U,
 `kLLWU_PinFilterFallingEdge` = 2U,
 `kLLWU_PinFilterAnyEdge` = 3U }
Digital filter control modes.

Enumeration Type Documentation

Driver version

- #define **FSL_LLWU_DRIVER_VERSION** (MAKE_VERSION(2, 0, 1))
LLWU driver version 2.0.1.

Low-Leakage Wakeup Unit Control APIs

- void **LLWU_SetExternalWakeUpPinMode** (LLWU_Type *base, uint32_t pinIndex, **llwu_external_pin_mode_t** pinMode)
Sets the external input pin source mode.
- bool **LLWU_GetExternalWakeUpPinFlag** (LLWU_Type *base, uint32_t pinIndex)
Gets the external wakeup source flag.
- void **LLWU_ClearExternalWakeUpPinFlag** (LLWU_Type *base, uint32_t pinIndex)
Clears the external wakeup source flag.
- static void **LLWU_EnableInternalModuleInterruptWakeup** (LLWU_Type *base, uint32_t moduleIndex, bool enable)
Enables/disables the internal module source.
- static bool **LLWU_GetInternalWakeUpModuleFlag** (LLWU_Type *base, uint32_t moduleIndex)
Gets the external wakeup source flag.
- void **LLWU_SetPinFilterMode** (LLWU_Type *base, uint32_t filterIndex, **llwu_external_pin_filter_mode_t** filterMode)
Sets the pin filter configuration.
- bool **LLWU_GetPinFilterFlag** (LLWU_Type *base, uint32_t filterIndex)
Gets the pin filter configuration.
- void **LLWU_ClearPinFilterFlag** (LLWU_Type *base, uint32_t filterIndex)
Clears the pin filter configuration.

21.5 Data Structure Documentation

21.5.1 struct **llwu_external_pin_filter_mode_t**

Data Fields

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

21.6 Macro Definition Documentation

21.6.1 #define **FSL_LLWU_DRIVER_VERSION** (MAKE_VERSION(2, 0, 1))

21.7 Enumeration Type Documentation

21.7.1 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.

21.7.2 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.

21.8 Function Documentation

21.8.1 void LLWU_SetExternalWakeupsPinMode (*LLWU_Type * base*, *uint32_t pinIndex*, *llwu_external_pin_mode_t pinMode*)

This function sets the external input pin source mode that is used as a wake up source.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>pinIndex</i>	A pin index to be enabled as an external wakeup source starting from 1.
<i>pinMode</i>	A pin configuration mode defined in the llwu_external_pin_modes_t.

21.8.2 bool LLWU_GetExternalWakeupsPinFlag (*LLWU_Type * base*, *uint32_t pinIndex*)

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>pinIndex</i>	A pin index, which starts from 1.

Returns

True if the specific pin is a wakeup source.

Function Documentation

21.8.3 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.

21.8.4 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

21.8.5 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.

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

This function sets the pin filter configuration.

Function Documentation

Parameters

<i>base</i>	LLWU peripheral base address.
<i>filterIndex</i>	A pin filter index used to enable/disable the digital filter, starting from 1.
<i>filterMode</i>	A filter mode configuration

21.8.7 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.

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

This function clears the pin filter flag.

Parameters

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

Chapter 22

LMEM: Local Memory Controller Cache Control Driver

22.1 Overview

The KSDK provides a peripheral driver for the Local Memory Controller Cache Controller module of Kinetis devices.

22.2 Descriptions

The LMEM Cache peripheral driver allows the user to enable/disable the cache and to perform cache maintenance operations such as invalidate, push, and clear. These maintenance operations may be performed on the Processor Code (PC) bus or Both Processor Code (PC) and Processor System (PS) bus.

The Kinetis devices contain a Processor Code (PC) bus and a Processor System (PS) bus as follows. The Processor Code (PC) bus - a 32-bit address space bus with low-order addresses (0x0000_0000 through 0x1FFF_FFFF) used normally for code access. The Processor System (PS) bus - a 32-bit address space bus with high-order addresses (0x2000_0000 through 0xFFFF_FFFF) used normally for data accesses.

Some Kinetic MCU devices have caches available for the PC bus and PS bus, others may only have a PC bus cache, while some do not have PC or PS caches at all. See the appropriate Kinetis reference manual for cache availability.

Cache maintenance operations:

command	description		
	Invalidate		U
	Push	P	ush a cache entry if it is valid and modified, then clear the m
	Clear	P	ush a cache entry if it is valid and modified, then clear the m

The above cache maintenance operations may be performed on the entire cache or on a line-basis. The peripheral driver API names distinguish between the two using the terms "All" or Line".

22.3 Function groups

22.3.1 Local Memory Processor Code Bus Cache Control

The invalidate command can be performed on the entire cache, one line, or multiple lines by calling [LMEM_CodeCacheInvalidateAll\(\)](#), [LMEM_CodeCacheInvalidateLine\(\)](#), and [LMEM_CodeCacheInvalidateMultiLines\(\)](#).

Function groups

The push command can be performed on the entire cache, one line, or multiple lines by calling [LMEM_CodeCachePushAll\(\)](#), [LMEM_CodeCachePushLine\(\)](#), and [LMEM_CodeCachePushMultiLines\(\)](#).

The clear command can be performed on the entire cache, one line, or multiple lines by calling [LMEM_CodeCacheClearAll\(\)](#), [LMEM_CodeCacheClearLine\(\)](#), and [LMEM_CodeCacheClearMultiLines\(\)](#).

Note that the parameter "address" must be supplied, which indicates the physical address of the line to perform the one line cache maintenance operation. In addition, the length of the number of bytes should be supplied for multiple line operation. The function determines if the length meets or exceeds 1/2 the cache size because the cache contains 2 WAYs, half of the cache is in WAY0 and the other half in WAY1 and if so, performs a cache maintenance "all" operation which is faster than performing the cache maintenance on a line-basis.

Cache Demotion: Cache region demotion - Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The cache region demote function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state. To demote a cache region, call the [LMEM_CodeCacheDemoteRegion\(\)](#).

Note that the address region assignment of the 16 subregions is device-specific and is detailed in the Chip Configuration part of the SoC Kinetis reference manual. The LMEM provides typedef enums for each of the 16 regions, starting with "kLMEM_CacheRegion0" and ending with "kLMEM_CacheRegion15". The parameter cacheMode is of type lmem_cache_mode_t. This provides typedef enums for each of the cache modes, such as "kLMEM_CacheNonCacheable", "kLMEM_CacheWriteThrough", and "kLMEM_CacheWriteBack". **Cache Enable and Disable:** The cache enable function enables the PC bus cache and the write buffer. However, before enabling these, the function first performs an invalidate all. Call [LMEM_EnableCodeCache\(\)](#) to enable a particular bus cache.

22.3.2 Local Memory Processor System Bus Cache Control

The invalidate command can be performed on the entire cache, one line, or multiple lines by calling [LMEM_SystemCacheInvalidateAll\(\)](#), [LMEM_SystemCacheInvalidateLine\(\)](#), and [LMEM_SystemCacheInvalidateMultiLines\(\)](#).

The push command can be performed on the entire cache, one line, or multiple lines by calling [LMEM_SystemCachePushAll\(\)](#), [LMEM_SystemCachePushLine\(\)](#), and [LMEM_SystemCachePushMultiLines\(\)](#).

The clear command can be performed on the entire cache, one line, or multiple lines by calling [LMEM_SystemCacheClearAll\(\)](#), [LMEM_SystemCacheClearLine\(\)](#), and [LMEM_SystemCacheClearMultiLines\(\)](#).

Note that the parameter "address" must be supplied, which indicates the physical address of the line to perform the one line cache maintenance operation. In addition, the length of the number of bytes should be supplied for multiple lines operation. The function determines if the length meets or exceeds 1/2 the cache size because the cache contains 2 WAYs, half of the cache is in WAY0 and the other half in WAY1 and if so, performs a cache maintenance "all" operation which is faster than performing the cache maintenance on a line-basis.

Cache Demotion: Cache region demotion - Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The cache region demote function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state. To demote a cache region, call the LMEM_SystemCacheDemoteRegion().

Note that the address region assignment of the 16 subregions is device-specific and is described in the Chip Configuration part of the Kinetis SoC reference manual. The LMEM provides typedef enumerations for each of the 16 regions, starting with "kLMEM_CacheRegion0" and ending with "kLMEM_CacheRegion15". The parameter cacheMode is of type lmem_cache_mode_t. This provides typedef enumerations for each of the cache modes, such as "kLMEM_CacheNonCacheable", "kLMEM_CacheWriteThrough", and "kLMEM_CacheWriteBack". Cache Enable and Disable: The cache enable function enables the PS bus cache and the write buffer. However, before enabling these, the function first performs an invalidate all. Call LMEM_EnableSystemCache() to enable a particular bus cache.

Macros

- #define **LMEM_CACHE_LINE_SIZE** (0x10U)
Cache line is 16-bytes.
- #define **LMEM_CACHE_SIZE_ONEWAY** (4096U)
Cache size is 4K-bytes one way.

Enumerations

- enum **lmem_cache_mode_t** {

kLMEM_NonCacheable = 0x0U,

kLMEM_CacheWriteThrough = 0x2U,

kLMEM_CacheWriteBack = 0x3U }

LMEM cache mode options.
- enum **lmem_cache_region_t** {

kLMEM_CacheRegion15 = 0U,

kLMEM_CacheRegion14,

kLMEM_CacheRegion13,

kLMEM_CacheRegion12,

kLMEM_CacheRegion11,

kLMEM_CacheRegion10,

kLMEM_CacheRegion9,

kLMEM_CacheRegion8,

kLMEM_CacheRegion7,

kLMEM_CacheRegion6,

kLMEM_CacheRegion5,

kLMEM_CacheRegion4,

kLMEM_CacheRegion3,

kLMEM_CacheRegion2,

kLMEM_CacheRegion1,

kLMEM_CacheRegion0 }

LMEM cache regions.

Macro Definition Documentation

- enum `lmem_cache_line_command_t` {
 `kLMEM_CacheLineSearchReadOrWrite` = 0U,
 `kLMEM_CacheLineInvalidate`,
 `kLMEM_CacheLinePush`,
 `kLMEM_CacheLineClear` }
 LMEM cache line command.

Driver version

- `#define FSL_LMEM_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`
 LMEM controller driver version 2.1.0.

Local Memory Processor Code Bus Cache Control

- void `LMEM_EnableCodeCache` (`LMEM_Type` *base, bool enable)
 Enables/disables the processor code bus cache.
- static void `LMEM_EnableCodeWriteBuffer` (`LMEM_Type` *base, bool enable)
 Enables/disables the processor code bus write buffer.
- void `LMEM_CodeCacheInvalidateAll` (`LMEM_Type` *base)
 Invalidates the processor code bus cache.
- void `LMEM_CodeCachePushAll` (`LMEM_Type` *base)
 Pushes all modified lines in the processor code bus cache.
- void `LMEM_CodeCacheClearAll` (`LMEM_Type` *base)
 Clears the processor code bus cache.
- void `LMEM_CodeCacheInvalidateLine` (`LMEM_Type` *base, `uint32_t` address)
 Invalidates a specific line in the processor code bus cache.
- void `LMEM_CodeCacheInvalidateMultiLines` (`LMEM_Type` *base, `uint32_t` address, `uint32_t` length)
 Invalidates multiple lines in the processor code bus cache.
- void `LMEM_CodeCachePushLine` (`LMEM_Type` *base, `uint32_t` address)
 Pushes a specific modified line in the processor code bus cache.
- void `LMEM_CodeCachePushMultiLines` (`LMEM_Type` *base, `uint32_t` address, `uint32_t` length)
 Pushes multiple modified lines in the processor code bus cache.
- void `LMEM_CodeCacheClearLine` (`LMEM_Type` *base, `uint32_t` address)
 Clears a specific line in the processor code bus cache.
- void `LMEM_CodeCacheClearMultiLines` (`LMEM_Type` *base, `uint32_t` address, `uint32_t` length)
 Clears multiple lines in the processor code bus cache.
- status_t `LMEM_CodeCacheDemoteRegion` (`LMEM_Type` *base, `lmem_cache_region_t` region, `lmem_cache_mode_t` cacheMode)
 Demotes the cache mode of a region in processor code bus cache.

22.4 Macro Definition Documentation

22.4.1 `#define FSL_LMEM_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`

22.4.2 `#define LMEM_CACHE_LINE_SIZE (0x10U)`

22.4.3 `#define LMEM_CACHE_SIZE_ONEWAY (4096U)`

22.5 Enumeration Type Documentation

22.5.1 enum lmem_cache_mode_t

Enumerator

- kLMEM_NonCacheable*** Cache mode: non-cacheable.
- kLMEM_CacheWriteThrough*** Cache mode: write-through.
- kLMEM_CacheWriteBack*** Cache mode: write-back.

22.5.2 enum lmem_cache_region_t

Enumerator

- kLMEM_CacheRegion15*** Cache Region 15.
- kLMEM_CacheRegion14*** Cache Region 14.
- kLMEM_CacheRegion13*** Cache Region 13.
- kLMEM_CacheRegion12*** Cache Region 12.
- kLMEM_CacheRegion11*** Cache Region 11.
- kLMEM_CacheRegion10*** Cache Region 10.
- kLMEM_CacheRegion9*** Cache Region 9.
- kLMEM_CacheRegion8*** Cache Region 8.
- kLMEM_CacheRegion7*** Cache Region 7.
- kLMEM_CacheRegion6*** Cache Region 6.
- kLMEM_CacheRegion5*** Cache Region 5.
- kLMEM_CacheRegion4*** Cache Region 4.
- kLMEM_CacheRegion3*** Cache Region 3.
- kLMEM_CacheRegion2*** Cache Region 2.
- kLMEM_CacheRegion1*** Cache Region 1.
- kLMEM_CacheRegion0*** Cache Region 0.

22.5.3 enum lmem_cache_line_command_t

Enumerator

- kLMEM_CacheLineSearchReadOrWrite*** Cache line search and read or write.
- kLMEM_CacheLineInvalidate*** Cache line invalidate.
- kLMEM_CacheLinePush*** Cache line push.
- kLMEM_CacheLineClear*** Cache line clear.

Function Documentation

22.6 Function Documentation

22.6.1 void LMEM_EnableCodeCache (LMEM_Type * *base*, bool *enable*)

This function enables/disables the cache. The function first invalidates the entire cache and then enables/disables both the cache and write buffers.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>enable</i>	The enable or disable flag. true - enable the code cache. false - disable the code cache.

22.6.2 static void LMEM_EnableCodeWriteBuffer (LMEM_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	LMEM peripheral base address.
<i>enable</i>	The enable or disable flag. true - enable the code bus write buffer. false - disable the code bus write buffer.

22.6.3 void LMEM_CodeCacheInvalidateAll (LMEM_Type * *base*)

This function invalidates the cache both ways, which means that it unconditionally clears valid bits and modifies bits of a cache entry.

Parameters

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

22.6.4 void LMEM_CodeCachePushAll (LMEM_Type * *base*)

This function pushes all modified lines in both ways in the entire cache. It pushes a cache entry if it is valid and modified and clears the modified bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

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

22.6.5 void LMEM_CodeCacheClearAll (LMEM_Type * *base*)

This function clears the entire cache and pushes (flushes) and invalidates the operation. Clear - Pushes a cache entry if it is valid and modified, then clears the valid and modified bits. If the entry is not valid or not modified, clear the valid bit.

Function Documentation

Parameters

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

22.6.6 void LMEM_CodeCacheInvalidateLine (LMEM_Type * *base*, uint32_t *address*)

This function invalidates a specific line in the cache based on the physical address passed in by the user. Invalidate - Unconditionally clears valid and modified bits of a cache entry.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>address</i>	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

22.6.7 void LMEM_CodeCacheInvalidateMultiLines (LMEM_Type * *base*, uint32_t *address*, uint32_t *length*)

This function invalidates multiple lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half the cache, the function performs an entire cache invalidate function, which is more efficient than invalidating the cache line-by-line. Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Invalidate - Unconditionally clear valid and modified bits of a cache entry.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>address</i>	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
<i>length</i>	The length in bytes of the total amount of cache lines.

22.6.8 void LMEM_CodeCachePushLine (LMEM_Type * *base*, uint32_t *address*)

This function pushes a specific modified line based on the physical address passed in by the user. Push - Push a cache entry if it is valid and modified, then clear the modified bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>address</i>	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

22.6.9 void LMEM_CodeCachePushMultiLines (LMEM_Type * *base*, uint32_t *address*, uint32_t *length*)

This function pushes multiple modified lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half of the cache, the function performs an cache push function, which is more efficient than pushing the modified lines in the cache line-by-line. Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Push - Push a cache entry if it is valid and modified, then clear the modified bit. If the entry is not valid or not modified, leave as is. This action does not clear the valid bit. A cache push is synonymous with a cache flush.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>address</i>	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
<i>length</i>	The length in bytes of the total amount of cache lines.

22.6.10 void LMEM_CodeCacheClearLine (LMEM_Type * *base*, uint32_t *address*)

This function clears a specific line based on the physical address passed in by the user. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If entry not valid or not modified, clear the valid bit.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>address</i>	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.

Function Documentation

22.6.11 void LMEM_CodeCacheClearMultiLines (**LMEM_Type * base, uint32_t address, uint32_t length**)

This function clears multiple lines in the cache based on the physical address and length in bytes passed in by the user. If the function detects that the length meets or exceeds half the total amount of cache, the function performs a cache clear function which is more efficient than clearing the lines in the cache line-by-line. Because the cache consists of two ways and line commands based on the physical address searches both ways, check half the total amount of cache. Clear - Push a cache entry if it is valid and modified, then clear the valid and modify bits. If entry not valid or not modified, clear the valid bit.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>address</i>	The physical address of the cache line. Should be 16-byte aligned address. If not, it is changed to the 16-byte aligned memory address.
<i>length</i>	The length in bytes of the total amount of cache lines.

22.6.12 status_t LMEM_CodeCacheDemoteRegion (**LMEM_Type * base, lmem_cache_region_t region, lmem_cache_mode_t cacheMode**)

This function allows the user to demote the cache mode of a region within the device's memory map. Demoting the cache mode reduces the cache function applied to a memory region from write-back to write-through to non-cacheable. The function checks to see if the requested cache mode is higher than or equal to the current cache mode, and if so, returns an error. After a region is demoted, its cache mode can only be raised by a reset, which returns it to its default state which is the highest cache configure for each region. To maintain cache coherency, changes to the cache mode should be completed while the address space being changed is not being accessed or the cache is disabled. Before a cache mode change, this function completes a cache clear all command to push and invalidate any cache entries that may have changed.

Parameters

<i>base</i>	LMEM peripheral base address.
<i>region</i>	The desired region to demote of type lmem_cache_region_t.
<i>cacheMode</i>	The new, demoted cache mode of type lmem_cache_mode_t.

Returns

The execution result. kStatus_Success The cache demote operation is successful. kStatus_Fail The cache demote operation is failure.

Chapter 23

LPTMR: Low-Power Timer

23.1 Overview

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

23.2 Function groups

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

23.2.1 Initialization and deinitialization

The function [LPTMR_Init\(\)](#) initializes the LPTMR with specified configurations. The function [LPTMR_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the LPTMR for a timer or a pulse counter mode mode. It also sets up the LPTMR's free running mode operation and a clock source.

The function [LPTMR_DeInit\(\)](#) disables the LPTMR module and gates the module clock.

23.2.2 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.

23.2.3 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

23.2.4 Status

Provides functions to get and clear the LPTMR status.

23.2.5 Interrupt

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

23.3 Typical use case

23.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically.

```
int main(void)
{
    uint32_t currentCounter = 0U;
    lptmr_config_t lptmrConfig;

    LED_INIT();

    /* Board pin, clock, debug console initialization */
    BOARD_InitHardware();

    /* Configures the LPTMR */
    LPTMR_GetDefaultConfig(&lptmrConfig);

    /* Initializes the LPTMR */
    LPTMR_Init(LPTMR0, &lptmrConfig);

    /* Sets the timer period */
    LPTMR_SetTimerPeriod(LPTMR0, USEC_TO_COUNT(1000000U, LPTMR_SOURCE_CLOCK));

    /* Enables a timer interrupt */
    LPTMR_EnableInterrupts(LPTMR0,
                           kLPTMR_TimerInterruptEnable);

    /* Enables the NVIC */
    EnableIRQ(LPTMR0_IRQn);

    PRINTF("Low Power Timer Example\r\n");

    /* Starts counting */
    LPTMR_StartTimer(LPTMR0);
    while (1)
    {
        if (currentCounter != lptmrCounter)
        {
            currentCounter = lptmrCounter;
            PRINTF("LPTMR interrupt No.%d \r\n", currentCounter);
        }
    }
}
```

Data Structures

- struct [lptmr_config_t](#)
LPTMR config structure. [More...](#)

Enumerations

- enum `lptmr_pin_select_t` {

 `kLPTMR_PinSelectInput_0` = 0x0U,

 `kLPTMR_PinSelectInput_1` = 0x1U,

 `kLPTMR_PinSelectInput_2` = 0x2U,

 `kLPTMR_PinSelectInput_3` = 0x3U }

LPTMR pin selection used in pulse counter mode.
- enum `lptmr_pin_polarity_t` {

 `kLPTMR_PinPolarityActiveHigh` = 0x0U,

 `kLPTMR_PinPolarityActiveLow` = 0x1U }

LPTMR pin polarity used in pulse counter mode.
- enum `lptmr_timer_mode_t` {

 `kLPTMR_TimerModeTimeCounter` = 0x0U,

 `kLPTMR_TimerModePulseCounter` = 0x1U }

LPTMR timer mode selection.
- enum `lptmr_prescaler_glitch_value_t` {

 `kLPTMR_Prescale_Glitch_0` = 0x0U,

 `kLPTMR_Prescale_Glitch_1` = 0x1U,

 `kLPTMR_Prescale_Glitch_2` = 0x2U,

 `kLPTMR_Prescale_Glitch_3` = 0x3U,

 `kLPTMR_Prescale_Glitch_4` = 0x4U,

 `kLPTMR_Prescale_Glitch_5` = 0x5U,

 `kLPTMR_Prescale_Glitch_6` = 0x6U,

 `kLPTMR_Prescale_Glitch_7` = 0x7U,

 `kLPTMR_Prescale_Glitch_8` = 0x8U,

 `kLPTMR_Prescale_Glitch_9` = 0x9U,

 `kLPTMR_Prescale_Glitch_10` = 0xAU,

 `kLPTMR_Prescale_Glitch_11` = 0xBU,

 `kLPTMR_Prescale_Glitch_12` = 0xCU,

 `kLPTMR_Prescale_Glitch_13` = 0xDU,

 `kLPTMR_Prescale_Glitch_14` = 0xEU,

 `kLPTMR_Prescale_Glitch_15` = 0xFU }

LPTMR prescaler/glitch filter values.
- enum `lptmr_prescaler_clock_select_t` {

 `kLPTMR_PrescalerClock_0` = 0x0U,

 `kLPTMR_PrescalerClock_1` = 0x1U,

 `kLPTMR_PrescalerClock_2` = 0x2U,

 `kLPTMR_PrescalerClock_3` = 0x3U }

LPTMR prescaler/glitch filter clock select.
- enum `lptmr_interrupt_enable_t` { `kLPTMR_TimerInterruptEnable` = LPTMR_CSR_TIE_MASK }

List of the LPTMR interrupts.
- enum `lptmr_status_flags_t` { `kLPTMR_TimerCompareFlag` = LPTMR_CSR_TCF_MASK }

List of the LPTMR status flags.

Driver version

- #define `FSL_LPTMR_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 0))

Version 2.0.0.

Initialization and deinitialization

- void [LPTMR_Init](#) (LPTMR_Type *base, const [lptmr_config_t](#) *config)
Ungates the LPTMR clock and configures the peripheral for a basic operation.
- void [LPTMR_Deinit](#) (LPTMR_Type *base)
Gates the LPTMR clock.
- void [LPTMR_GetDefaultConfig](#) ([lptmr_config_t](#) *config)
Fills in the LPTMR configuration structure with default settings.

Interrupt Interface

- static void [LPTMR_EnableInterrupts](#) (LPTMR_Type *base, uint32_t mask)
Enables the selected LPTMR interrupts.
- static void [LPTMR_DisableInterrupts](#) (LPTMR_Type *base, uint32_t mask)
Disables the selected LPTMR interrupts.
- static uint32_t [LPTMR_GetEnabledInterrupts](#) (LPTMR_Type *base)
Gets the enabled LPTMR interrupts.

Status Interface

- static uint32_t [LPTMR_GetStatusFlags](#) (LPTMR_Type *base)
Gets the LPTMR status flags.
- static void [LPTMR_ClearStatusFlags](#) (LPTMR_Type *base, uint32_t mask)
Clears the LPTMR status flags.

Read and write the timer period

- static void [LPTMR_SetTimerPeriod](#) (LPTMR_Type *base, uint16_t ticks)
Sets the timer period in units of count.
- static uint16_t [LPTMR_GetCurrentTimerCount](#) (LPTMR_Type *base)
Reads the current timer counting value.

Timer Start and Stop

- static void [LPTMR_StartTimer](#) (LPTMR_Type *base)
Starts the timer.
- static void [LPTMR_StopTimer](#) (LPTMR_Type *base)
Stops the timer.

23.4 Data Structure Documentation

23.4.1 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.

23.5 Enumeration Type Documentation

23.5.1 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.

23.5.2 enum lptmr_pin_polarity_t

Enumerator

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

23.5.3 enum lptmr_timer_mode_t

Enumerator

- `kLPTMR_TimerModeTimeCounter` Time Counter mode.
- `kLPTMR_TimerModePulseCounter` Pulse Counter mode.

Enumeration Type Documentation

23.5.4 enum lptmr_prescaler_glitch_value_t

Enumerator

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

23.5.5 enum lptmr_prescaler_clock_select_t

Note

Clock connections are SoC-specific

Enumerator

- kLPTMR_PrescalerClock_0*** Prescaler/glitch filter clock 0 selected.
- kLPTMR_PrescalerClock_1*** Prescaler/glitch filter clock 1 selected.
- kLPTMR_PrescalerClock_2*** Prescaler/glitch filter clock 2 selected.
- kLPTMR_PrescalerClock_3*** Prescaler/glitch filter clock 3 selected.

23.5.6 enum lptmr_interrupt_enable_t

Enumerator

- kLPTMR_TimerInterruptEnable*** Timer interrupt enable.

23.5.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

23.6 Function Documentation

23.6.1 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.

23.6.2 void LPTMR_Deinit (LPTMR_Type * *base*)

Parameters

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

23.6.3 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.
---------------	---

23.6.4 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

23.6.5 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 .

23.6.6 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](#)

23.6.7 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](#)

23.6.8 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 .

23.6.9 static void LPTMR_SetTimerPeriod (LPTMR_Type * *base*, uint16_t *ticks*) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note

1. The TCF flag is set with the CNR equals the count provided here and then increments.
2. Call the utility macros provided in the fsl_common.h to convert to ticks.

Parameters

<i>base</i>	LPTMR peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

23.6.10 static uint16_t LPTMR_GetCurrentTimerCount (LPTMR_Type * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Function Documentation

Note

Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

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

Returns

The current counter value in ticks

23.6.11 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
-------------	-------------------------------

23.6.12 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 24

LPUART: Low Power UART Driver

24.1 Overview

Modules

- LPUART DMA Driver
- LPUART Driver
- LPUART FreeRTOS Driver
- LPUART eDMA Driver
- LPUART µCOS/II Driver
- LPUART µCOS/III Driver

LPUART Driver

24.2 LPUART Driver

24.2.1 Overview

The KSDK provides a peripheral driver for the Low Power UART (LPUART) module of Kinetis devices.

24.2.2 Typical use case

24.2.2.1 LPUART Operation

```
uint8_t ch;
LPUART_GetDefaultConfig(&user_config);
user_config.baudRate = 115200U;
config.enableTx = true;
config.enableRx = true;

LPUART_Init(LPUART1, &user_config, 120000000U);

LPUART_WriteBlocking(LPUART1, txbuff, sizeof(txbuff) - 1);

while(1)
{
    LPUART_ReadBlocking(LPUART1, &ch, 1);
    LPUART_WriteBlocking(LPUART1, &ch, 1);
}
```

Data Structures

- struct [lpuart_config_t](#)
LPUART configuration structure. [More...](#)
- struct [lpuart_transfer_t](#)
LPUART transfer structure. [More...](#)
- struct [lpuart_handle_t](#)
LPUART handle structure. [More...](#)

Typedefs

- typedef void(* [lpuart_transfer_callback_t](#))(LPUART_Type *base, lpuart_handle_t *handle, status_t status, void *userData)
LPUART transfer callback function.

Enumerations

- enum `_lpuart_status` {

kStatus_LPUART_TxBusy = MAKE_STATUS(kStatusGroup_LPUART, 0),

kStatus_LPUART_RxBusy = MAKE_STATUS(kStatusGroup_LPUART, 1),

kStatus_LPUART_TxIdle = MAKE_STATUS(kStatusGroup_LPUART, 2),

kStatus_LPUART_RxIdle = MAKE_STATUS(kStatusGroup_LPUART, 3),

kStatus_LPUART_TxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_LPUART, 4),

kStatus_LPUART_RxWatermarkTooLarge = MAKE_STATUS(kStatusGroup_LPUART, 5),

kStatus_LPUART_FlagCannotClearManually = MAKE_STATUS(kStatusGroup_LPUART, 6),

kStatus_LPUART_Error = MAKE_STATUS(kStatusGroup_LPUART, 7),

kStatus_LPUART_RxRingBufferOverrun,

kStatus_LPUART_RxHardwareOverrun = MAKE_STATUS(kStatusGroup_LPUART, 9),

kStatus_LPUART_NoiseError = MAKE_STATUS(kStatusGroup_LPUART, 10),

kStatus_LPUART_FramingError = MAKE_STATUS(kStatusGroup_LPUART, 11),

kStatus_LPUART_ParityError = MAKE_STATUS(kStatusGroup_LPUART, 12),

kStatus_LPUART_BaudrateNotSupport }

Error codes for the LPUART driver.
- enum `lpuart_parity_mode_t` {

kLPUART_ParityDisabled = 0x0U,

kLPUART_ParityEven = 0x2U,

kLPUART_ParityOdd = 0x3U }

LPUART parity mode.
- enum `lpuart_data_bits_t` { kLPUART_EightDataBits = 0x0U }

LPUART data bits count.
- enum `lpuart_stop_bit_count_t` {

kLPUART_OneStopBit = 0U,

kLPUART_TwoStopBit = 1U }

LPUART stop bit count.
- enum `_lpuart_interrupt_enable` {

kLPUART_LinBreakInterruptEnable = (LPUART_BAUD_LBKDIIE_MASK >> 8),

kLPUART_RxActiveEdgeInterruptEnable = (LPUART_BAUD_RXEDGIE_MASK >> 8),

kLPUART_TxDataRegEmptyInterruptEnable = (LPUART_CTRL_TIE_MASK),

kLPUART_TransmissionCompleteInterruptEnable = (LPUART_CTRL_TCIE_MASK),

kLPUART_RxDataRegFullInterruptEnable = (LPUART_CTRL_RIE_MASK),

kLPUART_IdleLineInterruptEnable = (LPUART_CTRL_ILIE_MASK),

kLPUART_RxOverrunInterruptEnable = (LPUART_CTRL_ORIE_MASK),

kLPUART_NoiseErrorInterruptEnable = (LPUART_CTRL_NEIE_MASK),

kLPUART_FramingErrorInterruptEnable = (LPUART_CTRL_FEIE_MASK),

kLPUART_ParityErrorInterruptEnable = (LPUART_CTRL_PEIE_MASK) }

LPUART interrupt configuration structure, default settings all disabled.
- enum `_lpuart_flags` {

LPUART Driver

```
kLPUART_TxDataRegEmptyFlag,  
kLPUART_TransmissionCompleteFlag,  
kLPUART_RxDataRegFullFlag,  
kLPUART_IdleLineFlag = (LPUART_STAT_IDLE_MASK),  
kLPUART_RxOverrunFlag = (LPUART_STAT_OR_MASK),  
kLPUART_NoiseErrorFlag = (LPUART_STAT_NF_MASK),  
kLPUART_FramingErrorFlag,  
kLPUART_ParityErrorFlag = (LPUART_STAT_PF_MASK),  
kLPUART_LinBreakFlag = (LPUART_STAT_LBKDIF_MASK),  
kLPUART_RxActiveEdgeFlag,  
kLPUART_RxActiveFlag,  
kLPUART_DataMatch1Flag = LPUART_STAT_MA1F_MASK,  
kLPUART_DataMatch2Flag = LPUART_STAT_MA2F_MASK,  
kLPUART_NoiseErrorInRxDataRegFlag,  
kLPUART_ParityErrorInRxDataRegFlag }
```

LPUART status flags.

Driver version

- #define **FSL_LPUART_DRIVER_VERSION** (MAKE_VERSION(2, 2, 3))
LPUART driver version 2.2.1.

Initialization and deinitialization

- status_t **LPUART_Init** (LPUART_Type *base, const lpuart_config_t *config, uint32_t srcClock_Hz)
Initializes an LPUART instance with the user configuration structure and the peripheral clock.
- void **LPUART_Deinit** (LPUART_Type *base)
Deinitializes a LPUART instance.
- void **LPUART_GetDefaultConfig** (lpuart_config_t *config)
Gets the default configuration structure.
- status_t **LPUART_SetBaudRate** (LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the LPUART instance baudrate.

Status

- uint32_t **LPUART_GetStatusFlags** (LPUART_Type *base)
Gets LPUART status flags.
- status_t **LPUART_ClearStatusFlags** (LPUART_Type *base, uint32_t mask)
Clears status flags with a provided mask.

Interrupts

- void [LPUART_EnableInterrupts](#) (LPUART_Type *base, uint32_t mask)
Enables LPUART interrupts according to a provided mask.
- void [LPUART_DisableInterrupts](#) (LPUART_Type *base, uint32_t mask)
Disables LPUART interrupts according to a provided mask.
- uint32_t [LPUART_GetEnabledInterrupts](#) (LPUART_Type *base)
Gets enabled LPUART interrupts.
- static uint32_t [LPUART_GetDataRegisterAddress](#) (LPUART_Type *base)
Gets the LPUART data register address.
- static void [LPUART_EnableTxDMA](#) (LPUART_Type *base, bool enable)
Enables or disables the LPUART transmitter DMA request.
- static void [LPUART_EnableRxDMA](#) (LPUART_Type *base, bool enable)
Enables or disables the LPUART receiver DMA.

Bus Operations

- static void [LPUART_EnableTx](#) (LPUART_Type *base, bool enable)
Enables or disables the LPUART transmitter.
- static void [LPUART_EnableRx](#) (LPUART_Type *base, bool enable)
Enables or disables the LPUART receiver.
- static void [LPUART_WriteByte](#) (LPUART_Type *base, uint8_t data)
Writes to the transmitter register.
- static uint8_t [LPUART_ReadByte](#) (LPUART_Type *base)
Reads the receiver register.
- void [LPUART_WriteBlocking](#) (LPUART_Type *base, const uint8_t *data, size_t length)
Writes to the transmitter register using a blocking method.
- status_t [LPUART_ReadBlocking](#) (LPUART_Type *base, uint8_t *data, size_t length)
Reads the receiver data register using a blocking method.

Transactional

- void [LPUART_TransferCreateHandle](#) (LPUART_Type *base, lpuart_handle_t *handle, [lpuart_transfer_callback_t](#) callback, void *userData)
Initializes the LPUART handle.
- status_t [LPUART_TransferSendNonBlocking](#) (LPUART_Type *base, lpuart_handle_t *handle, [lpuart_transfer_t](#) *xfer)
Transmits a buffer of data using the interrupt method.
- void [LPUART_TransferStartRingBuffer](#) (LPUART_Type *base, lpuart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
Sets up the RX ring buffer.
- void [LPUART_TransferStopRingBuffer](#) (LPUART_Type *base, lpuart_handle_t *handle)
Aborts the background transfer and uninstalls the ring buffer.
- void [LPUART_TransferAbortSend](#) (LPUART_Type *base, lpuart_handle_t *handle)
Aborts the interrupt-driven data transmit.
- status_t [LPUART_TransferGetSendCount](#) (LPUART_Type *base, lpuart_handle_t *handle, uint32_t *count)
Gets the number of bytes that have been written to the LPUART transmitter register.

LPUART Driver

- status_t [LPUART_TransferReceiveNonBlocking](#) (LPUART_Type *base, lpuart_handle_t *handle, lpuart_transfer_t *xfer, size_t *receivedBytes)
Receives a buffer of data using the interrupt method.
- void [LPUART_TransferAbortReceive](#) (LPUART_Type *base, lpuart_handle_t *handle)
Aborts the interrupt-driven data receiving.
- status_t [LPUART_TransferGetReceiveCount](#) (LPUART_Type *base, lpuart_handle_t *handle, uint32_t *count)
Gets the number of bytes that have been received.
- void [LPUART_TransferHandleIRQ](#) (LPUART_Type *base, lpuart_handle_t *handle)
LPUART IRQ handle function.
- void [LPUART_TransferHandleErrorIRQ](#) (LPUART_Type *base, lpuart_handle_t *handle)
LPUART Error IRQ handle function.

24.2.3 Data Structure Documentation

24.2.3.1 struct lpuart_config_t

Data Fields

- uint32_t **baudRate_Bps**
LPUART baud rate.
- lpuart_parity_mode_t **parityMode**
Parity mode, disabled (default), even, odd.
- lpuart_data_bits_t **dataBitsCount**
Data bits count, eight (default), seven.
- bool **isMsb**
Data bits order, LSB (default), MSB.
- lpuart_stop_bit_count_t **stopBitCount**
Number of stop bits, 1 stop bit (default) or 2 stop bits.
- bool **enableTx**
Enable TX.
- bool **enableRx**
Enable RX.

24.2.3.2 struct lpuart_transfer_t

Data Fields

- uint8_t * **data**
The buffer of data to be transfer.
- size_t **dataSize**
The byte count to be transfer.

24.2.3.2.0.65 Field Documentation

24.2.3.2.0.65.1 `uint8_t* lpuart_transfer_t::data`

24.2.3.2.0.65.2 `size_t lpuart_transfer_t::dataSize`

24.2.3.3 struct _lpuart_handle

Data Fields

- `uint8_t *volatile txData`
Address of remaining data to send.
- `volatile size_t txDataSize`
Size of the remaining data to send.
- `size_t txDataSizeAll`
Size of the data to send out.
- `uint8_t *volatile rxData`
Address of remaining data to receive.
- `volatile size_t rxDataSize`
Size of the remaining data to receive.
- `size_t rxDataSizeAll`
Size of the data to receive.
- `uint8_t * rxRingBuffer`
Start address of the receiver ring buffer.
- `size_t rxRingBufferSize`
Size of the ring buffer.
- `volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.
- `volatile uint16_t rxRingBufferTail`
Index for the user to get data from the ring buffer.
- `lpuart_transfer_callback_t callback`
Callback function.
- `void * userData`
LPUART callback function parameter.
- `volatile uint8_t txState`
TX transfer state.
- `volatile uint8_t rxState`
RX transfer state.

LPUART Driver

24.2.3.3.0.66 Field Documentation

- 24.2.3.3.0.66.1 `uint8_t* volatile Ipuart_handle_t::txData`
- 24.2.3.3.0.66.2 `volatile size_t Ipuart_handle_t::txDataSize`
- 24.2.3.3.0.66.3 `size_t Ipuart_handle_t::txDataSizeAll`
- 24.2.3.3.0.66.4 `uint8_t* volatile Ipuart_handle_t::rxData`
- 24.2.3.3.0.66.5 `volatile size_t Ipuart_handle_t::rxDataSize`
- 24.2.3.3.0.66.6 `size_t Ipuart_handle_t::rxDataSizeAll`
- 24.2.3.3.0.66.7 `uint8_t* Ipuart_handle_t::rxRingBuffer`
- 24.2.3.3.0.66.8 `size_t Ipuart_handle_t::rxRingBufferSize`
- 24.2.3.3.0.66.9 `volatile uint16_t Ipuart_handle_t::rxRingBufferHead`
- 24.2.3.3.0.66.10 `volatile uint16_t Ipuart_handle_t::rxRingBufferTail`
- 24.2.3.3.0.66.11 `Ipuart_transfer_callback_t Ipuart_handle_t::callback`
- 24.2.3.3.0.66.12 `void* Ipuart_handle_t::userData`
- 24.2.3.3.0.66.13 `volatile uint8_t Ipuart_handle_t::txState`
- 24.2.3.3.0.66.14 `volatile uint8_t Ipuart_handle_t::rxState`

24.2.4 Macro Definition Documentation

- 24.2.4.1 `#define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))`

24.2.5 Typedef Documentation

- 24.2.5.1 `typedef void(* Ipuart_transfer_callback_t)(LPUART_Type *base, Ipuart_handle_t *handle, status_t status, void *userData)`

24.2.6 Enumeration Type Documentation

24.2.6.1 enum _Ipuart_status

Enumerator

- `kStatus_LPUART_TxBusy` TX busy.
- `kStatus_LPUART_RxBusy` RX busy.
- `kStatus_LPUART_TxIdle` LPUART transmitter is idle.

kStatus_LPUART_RxIdle LPUART receiver is idle.
kStatus_LPUART_TxWatermarkTooLarge TX FIFO watermark too large.
kStatus_LPUART_RxWatermarkTooLarge RX FIFO watermark too large.
kStatus_LPUART_FlagCannotClearManually Some flag can't manually clear.
kStatus_LPUART_Error Error happens on LPUART.
kStatus_LPUART_RxRingBufferOverrun LPUART RX software ring buffer overrun.
kStatus_LPUART_RxHardwareOverrun LPUART RX receiver overrun.
kStatus_LPUART_NoiseError LPUART noise error.
kStatus_LPUART_FramingError LPUART framing error.
kStatus_LPUART_ParityError LPUART parity error.
kStatus_LPUART_BaudrateNotSupport Baudrate is not support in current clock source.

24.2.6.2 enum lpuart_parity_mode_t

Enumerator

kLPUART_ParityDisabled Parity disabled.
kLPUART_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kLPUART_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

24.2.6.3 enum lpuart_data_bits_t

Enumerator

kLPUART_EightDataBits Eight data bit.

24.2.6.4 enum lpuart_stop_bit_count_t

Enumerator

kLPUART_OneStopBit One stop bit.
kLPUART_TwoStopBit Two stop bits.

24.2.6.5 enum _lpuart_interrupt_enable

This structure contains the settings for all LPUART interrupt configurations.

Enumerator

kLPUART_LinBreakInterruptEnable LIN break detect.
kLPUART_RxActiveEdgeInterruptEnable Receive Active Edge.
kLPUART_TxDataRegEmptyInterruptEnable Transmit data register empty.

kLPUART_TransmissionCompleteInterruptEnable Transmission complete.
kLPUART_RxDataRegFullInterruptEnable Receiver data register full.
kLPUART_IdleLineInterruptEnable Idle line.
kLPUART_RxOverrunInterruptEnable Receiver Overrun.
kLPUART_NoiseErrorInterruptEnable Noise error flag.
kLPUART_FramingErrorInterruptEnable Framing error flag.
kLPUART_ParityErrorInterruptEnable Parity error flag.

24.2.6.6 enum _lpuart_flags

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

Enumerator

kLPUART_TxDataRegEmptyFlag Transmit data register empty flag, sets when transmit buffer is empty.
kLPUART_TransmissionCompleteFlag Transmission complete flag, sets when transmission activity complete.
kLPUART_RxDataRegFullFlag Receive data register full flag, sets when the receive data buffer is full.
kLPUART_IdleLineFlag Idle line detect flag, sets when idle line detected.
kLPUART_RxOverrunFlag Receive Overrun, sets when new data is received before data is read from receive register.
kLPUART_NoiseErrorFlag Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kLPUART_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected.
kLPUART_ParityErrorFlag If parity enabled, sets upon parity error detection.
kLPUART_LinBreakFlag LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.
kLPUART_RxActiveEdgeFlag Receive pin active edge interrupt flag, sets when active edge detected.
kLPUART_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start bit.
kLPUART_DataMatch1Flag The next character to be read from LPUART_DATA matches MA1.
kLPUART_DataMatch2Flag The next character to be read from LPUART_DATA matches MA2.
kLPUART_NoiseErrorInRxDataRegFlag NOISY bit, sets if noise detected in current data word.
kLPUART_ParityErrorInRxDataRegFlag PARITYE bit, sets if noise detected in current data word.

24.2.7 Function Documentation

24.2.7.1 status_t LPUART_Init (LPUART_Type * *base*, const lpuart_config_t * *config*, uint32_t *srcClock_Hz*)

This function configures the LPUART module with user-defined settings. Call the [LPUART_GetDefaultConfig\(\)](#) function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
* lpuart_config_t lpuartConfig;
* lpuartConfig.baudRate_Bps = 115200U;
* lpuartConfig.parityMode = kLPUART_ParityDisabled;
* lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
* lpuartConfig.isMsb = false;
* lpuartConfig.stopBitCount = kLPUART_OneStopBit;
* lpuartConfig.txFifoWatermark = 0;
* lpuartConfig.rxFifoWatermark = 1;
* LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
*
```

Parameters

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

Return values

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

24.2.7.2 void LPUART_Deinit (LPUART_Type * *base*)

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

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

24.2.7.3 void LPUART_GetDefaultConfig (lpuart_config_t * *config*)

This function initializes the LPUART configuration structure to a default value. The default values are:
: lpuartConfig->baudRate_Bps = 115200U; lpuartConfig->parityMode = kLPUART_ParityDisabled;

LPUART Driver

```
lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;
```

Parameters

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

24.2.7.4 status_t LPUART_SetBaudRate (LPUART_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART_Init.

```
*   LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
*
```

Parameters

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

Return values

<i>kStatus_LPUART_BaudrateNotSupport</i>	Baudrate is not supported in the current clock source.
<i>kStatus_Success</i>	Set baudrate succeeded.

24.2.7.5 uint32_t LPUART_GetStatusFlags (LPUART_Type * *base*)

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators [_lpuart_flags](#). To check for a specific status, compare the return value with enumerators in the [_lpuart_flags](#). For example, to check whether the TX is empty:

```
*     if (kLPUART_TxDataRegEmptyFlag &
*         LPUART_GetStatusFlags(LPUART1))
*     {
*         ...
*     }
*
```

LPUART Driver

Parameters

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

Returns

LPUART status flags which are ORed by the enumerators in the _lpuart_flags.

24.2.7.6 **status_t LPUART_ClearStatusFlags (LPUART_Type * *base*, uint32_t *mask*)**

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only be cleared or set by hardware are: kLPUART_TxDataRegEmptyFlag, kLPUART_TransmissionCompleteFlag, kLPUART_RxDataRegFullFlag, kLPUART_RxActiveFlag, kLPUART_NoiseErrorInRxDataRegFlag, kLPUART_ParityErrorInRxDataRegFlag, kLPUART_TxFifoEmptyFlag, kLPUART_RxFifoEmptyFlag. Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>mask</i>	the status flags to be cleared. The user can use the enumerators in the _lpuart_status_flag_t to do the OR operation and get the mask.

Returns

0 succeed, others failed.

Return values

<i>kStatus_LPUART_FlagCannotClearManually</i>	The flag can't be cleared by this function but it is cleared automatically by hardware.
<i>kStatus_Success</i>	Status in the mask are cleared.

24.2.7.7 **void LPUART_EnableInterrupts (LPUART_Type * *base*, uint32_t *mask*)**

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the [_lpuart_interrupt_enable](#). This examples shows how to enable TX empty interrupt and RX full interrupt:

```
*      LPUART_EnableInterrupts(LPUART1,
*                                kLPUART_TxDataRegEmptyInterruptEnable |
*                                kLPUART_RxDataRegFullInterruptEnable);
*
```

Parameters

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

24.2.7.8 void LPUART_DisableInterrupts (LPUART_Type * *base*, uint32_t *mask*)

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See [_lpuart_interrupt_enable](#). This example shows how to disable the TX empty interrupt and RX full interrupt:

```
*     LPUART_DisableInterrupts(LPUART1,
*                               kLPUART_TxDataRegEmptyInterruptEnable |
*                               kLPUART_RxDataRegFullInterruptEnable);
*
```

Parameters

<i>base</i>	LPUART peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of _lpuart_interrupt_enable .

24.2.7.9 uint32_t LPUART_GetEnabledInterrupts (LPUART_Type * *base*)

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators [_lpuart_interrupt_enable](#). To check a specific interrupt enable status, compare the return value with enumerators in [_lpuart_interrupt_enable](#). For example, to check whether the TX empty interrupt is enabled:

```
*     uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);
*
*     if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
*     {
*         ...
*     }
*
```

Parameters

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

Returns

LPUART interrupt flags which are logical OR of the enumerators in [_lpuart_interrupt_enable](#).

24.2.7.10 static uint32_t LPUART_GetDataRegisterAddress (LPUART_Type * *base*) [inline], [static]

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

Parameters

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

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

24.2.7.11 static void LPUART_EnableTxDMA (LPUART_Type * *base*, bool *enable*) [inline], [static]

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

Parameters

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

24.2.7.12 static void LPUART_EnableRxDMA (LPUART_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the receiver data register full flag, STAT[RDRF], to generate DMA requests.

Parameters

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

24.2.7.13 static void LPUART_EnableTx (LPUART_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the LPUART transmitter.

Parameters

LPUART Driver

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

**24.2.7.14 static void LPUART_EnableRx (LPUART_Type * *base*, bool *enable*)
[inline], [static]**

This function enables or disables the LPUART receiver.

Parameters

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

**24.2.7.15 static void LPUART_WriteByte (LPUART_Type * *base*, uint8_t *data*)
[inline], [static]**

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

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

**24.2.7.16 static uint8_t LPUART_ReadByte (LPUART_Type * *base*) [inline],
[static]**

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

Parameters

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

Returns

Data read from data register.

24.2.7.17 void LPUART_WriteBlocking (LPUART_Type * *base*, const uint8_t * *data*, size_t *length*)

This function polls the transmitter register, waits for the register to be empty or for TX FIFO to have room, and writes data to the transmitter buffer.

Note

This function does not check whether all data has been sent out to the bus. Before disabling the transmitter, check the kLPUART_TransmissionCompleteFlag to ensure that the transmit is finished.

Parameters

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

24.2.7.18 status_t LPUART_ReadBlocking (LPUART_Type * *base*, uint8_t * *data*, size_t *length*)

This function polls the receiver register, waits for the receiver register full or receiver FIFO has data, and reads data from the TX register.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>data</i>	Start address of the buffer to store the received data.
<i>length</i>	Size of the buffer.

Return values

<i>kStatus_LPUART_Rx-HardwareOverrun</i>	Receiver overrun happened while receiving data.
<i>kStatus_LPUART_Noise-Error</i>	Noise error happened while receiving data.
<i>kStatus_LPUART_FramingError</i>	Framing error happened while receiving data.
<i>kStatus_LPUART_Parity-Error</i>	Parity error happened while receiving data.
<i>kStatus_Success</i>	Successfully received all data.

LPUART Driver

24.2.7.19 void LPUART_TransferCreateHandle (LPUART_Type * *base*, Ipuart_handle_t * *handle*, Ipuart_transfer_callback_t *callback*, void * *userData*)

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

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

Parameters

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

24.2.7.20 status_t LPUART_TransferSendNonBlocking (LPUART_Type * *base*, Ipuart_handle_t * *handle*, Ipuart_transfer_t * *xfer*)

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the [kStatus_LPUART_TxIdle](#) as status parameter.

Note

The [kStatus_LPUART_TxIdle](#) is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the T-X, check the [kLPUART_TransmissionCompleteFlag](#) to ensure that the transmit is finished.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>xfer</i>	LPUART transfer structure, see Ipuart_transfer_t .

Return values

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

24.2.7.21 void LPUART_TransferStartRingBuffer (LPUART_Type * *base*, Ipuart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

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

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

Note

When using RX ring buffer, one byte is reserved for internal use. In other words, if *ringBufferSize* is 32, then only 31 bytes are used for saving data.

Parameters

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

24.2.7.22 void LPUART_TransferStopRingBuffer (LPUART_Type * *base*, Ipuart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

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

LPUART Driver

24.2.7.23 void LPUART_TransferAbortSend (LPUART_Type * *base*, Ipuart_handle_t * *handle*)

This function aborts the interrupt driven data sending. The user can get the remainBtyes to find out how many bytes are not sent out.

Parameters

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

24.2.7.24 status_t LPUART_TransferGetSendCount (**LPUART_Type * base,** **Ipuart_handle_t * handle, uint32_t * count**)

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

Parameters

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

Return values

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

24.2.7.25 status_t LPUART_TransferReceiveNonBlocking (**LPUART_Type * base,** **Ipuart_handle_t * handle, Ipuart_transfer_t * xfer, size_t * receivedBytes**)

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter *receivedBytes* shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter *kStatus_UART_RxIdle*. For example, the upper layer needs 10 bytes but there are only 5 bytes in ring buffer. The 5 bytes are copied to *xfer->data*, which returns with the parameter *receivedBytes* set to 5. For the remaining 5 bytes, the newly arrived data is saved from *xfer->data[5]*. When 5 bytes are received, the LPUART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to *xfer->data*. When all data is received, the upper layer is notified.

LPUART Driver

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>xfer</i>	LPUART transfer structure, see uart_transfer_t .
<i>receivedBytes</i>	Bytes received from the ring buffer directly.

Return values

<i>kStatus_Success</i>	Successfully queue the transfer into the transmit queue.
<i>kStatus_LPUART_Rx-Busy</i>	Previous receive request is not finished.
<i>kStatus_InvalidArgument</i>	Invalid argument.

24.2.7.26 void LPUART_TransferAbortReceive (LPUART_Type * *base*, Ipuart_handle_t * *handle*)

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to find out how many bytes not received yet.

Parameters

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

24.2.7.27 status_t LPUART_TransferGetReceiveCount (LPUART_Type * *base*, Ipuart_handle_t * *handle*, uint32_t * *count*)

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

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>count</i>	Receive bytes count.

Return values

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

24.2.7.28 void LPUART_TransferHandleIRQ (LPUART_Type * *base*, Ipuart_handle_t * *handle*)

This function handles the LPUART transmit and receive IRQ request.

Parameters

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

24.2.7.29 void LPUART_TransferHandleErrorIRQ (LPUART_Type * *base*, Ipuart_handle_t * *handle*)

This function handles the LPUART error IRQ request.

Parameters

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

24.3 LPUART DMA Driver

24.3.1 Overview

Data Structures

- struct [lpuart_dma_handle_t](#)
LPUART DMA handle. [More...](#)

TypeDefs

- typedef void(* [lpuart_dma_transfer_callback_t](#))[\(LPUART_Type *base, lpuart_dma_handle_t *handle, status_t status, void *userData\)](#)
LPUART transfer callback function.

EDMA transactional

- void [LPUART_TransferCreateHandleDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle, [lpuart_dma_transfer_callback_t](#) callback, void *userData, dma_handle_t *txDmaHandle, dma_handle_t *rxDmaHandle)
Initializes the LPUART handle which is used in transactional functions.
- status_t [LPUART_TransferSendDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle, [lpuart_transfer_t](#) *xfer)
Sends data using DMA.
- status_t [LPUART_TransferReceiveDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle, [lpuart_transfer_t](#) *xfer)
Receives data using DMA.
- void [LPUART_TransferAbortSendDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle)
Aborts the sent data using DMA.
- void [LPUART_TransferAbortReceiveDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle)
Aborts the received data using DMA.
- status_t [LPUART_TransferGetSendCountDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle, uint32_t *count)
Gets the number of bytes written to the LPUART TX register.
- status_t [LPUART_TransferGetReceiveCountDMA](#) (LPUART_Type *base, lpuart_dma_handle_t *handle, uint32_t *count)
Gets the number of received bytes.

24.3.2 Data Structure Documentation

24.3.2.1 struct _lpuart_dma_handle

Data Fields

- [lpuart_dma_transfer_callback_t](#) [callback](#)

- *Callback function.*
- `void * userData`
LPUART callback function parameter.
- `size_t rxDataSizeAll`
Size of the data to receive.
- `size_t txDataSizeAll`
Size of the data to send out.
- `dma_handle_t * txDmaHandle`
The DMA TX channel used.
- `dma_handle_t * rxDmaHandle`
The DMA RX channel used.
- `volatile uint8_t txState`
TX transfer state.
- `volatile uint8_t rxState`
RX transfer state.

24.3.2.1.0.67 Field Documentation

24.3.2.1.0.67.1 `lpuart_dma_transfer_callback_t lpuart_dma_handle_t::callback`

24.3.2.1.0.67.2 `void* lpuart_dma_handle_t::userData`

24.3.2.1.0.67.3 `size_t lpuart_dma_handle_t::rxDataSizeAll`

24.3.2.1.0.67.4 `size_t lpuart_dma_handle_t::txDataSizeAll`

24.3.2.1.0.67.5 `dma_handle_t* lpuart_dma_handle_t::txDmaHandle`

24.3.2.1.0.67.6 `dma_handle_t* lpuart_dma_handle_t::rxDmaHandle`

24.3.2.1.0.67.7 `volatile uint8_t lpuart_dma_handle_t::txState`

24.3.3 Typedef Documentation

24.3.3.1 `typedef void(* lpuart_dma_transfer_callback_t)(LPUART_Type *base,
lpuart_dma_handle_t *handle, status_t status, void *userData)`

24.3.4 Function Documentation

24.3.4.1 `void LPUART_TransferCreateHandleDMA (LPUART_Type * base,
lpuart_dma_handle_t * handle, lpuart_dma_transfer_callback_t callback, void *
userData, dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle)`

LPUART DMA Driver

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to lpuart_dma_handle_t structure.
<i>callback</i>	Callback function.
<i>userData</i>	User data.
<i>txDmaHandle</i>	User-requested DMA handle for TX DMA transfer.
<i>rxDmaHandle</i>	User-requested DMA handle for RX DMA transfer.

24.3.4.2 status_t LPUART_TransferSendDMA (**LPUART_Type** * *base*, **lpuart_dma_handle_t** * *handle*, **lpuart_transfer_t** * *xfer*)

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

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>xfer</i>	LPUART DMA transfer structure. See lpuart_transfer_t .

Return values

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

24.3.4.3 status_t LPUART_TransferReceiveDMA (**LPUART_Type** * *base*, **lpuart_dma_handle_t** * *handle*, **lpuart_transfer_t** * *xfer*)

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

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to lpuart_dma_handle_t structure.
<i>xfer</i>	LPUART DMA transfer structure. See lpuart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeed, others failed.
<i>kStatus_LPUART_Rx-Busy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

24.3.4.4 void LPUART_TransferAbortSendDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*)

This function aborts send data using DMA.

Parameters

<i>base</i>	LPUART peripheral base address
<i>handle</i>	Pointer to lpuart_dma_handle_t structure

24.3.4.5 void LPUART_TransferAbortReceiveDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*)

This function aborts the received data using DMA.

Parameters

<i>base</i>	LPUART peripheral base address
<i>handle</i>	Pointer to lpuart_dma_handle_t structure

24.3.4.6 status_t LPUART_TransferGetSendCountDMA (LPUART_Type * *base*, lpuart_dma_handle_t * *handle*, uint32_t * *count*)

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

LPUART DMA Driver

Parameters

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

Return values

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

24.3.4.7 **status_t LPUART_TransferGetReceiveCountDMA (LPUART_Type * *base*, Ipuart_dma_handle_t * *handle*, uint32_t * *count*)**

This function gets the number of received bytes.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>count</i>	Receive bytes count.

Return values

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

24.4 LPUART eDMA Driver

24.4.1 Overview

Data Structures

- struct [lpuart_edma_handle_t](#)
LPUART eDMA handle. [More...](#)

TypeDefs

- [typedef void\(* lpuart_edma_transfer_callback_t \)](#)(LPUART_Type *base, lpuart_edma_handle_t *handle, status_t status, void *userData)
LPUART transfer callback function.

eDMA transactional

- void [LPUART_TransferCreateHandleEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle, [lpuart_edma_transfer_callback_t](#) callback, void *userData, [edma_handle_t](#) *txEdmaHandle, [edma_handle_t](#) *rxEdmaHandle)
Initializes the LPUART handle which is used in transactional functions.
- status_t [LPUART_SendEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle, [lpuart_transfer_t](#) *xfer)
Sends data using eDMA.
- status_t [LPUART_ReceiveEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle, [lpuart_transfer_t](#) *xfer)
Receives data using eDMA.
- void [LPUART_TransferAbortSendEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle)
Aborts the sent data using eDMA.
- void [LPUART_TransferAbortReceiveEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle)
Aborts the received data using eDMA.
- status_t [LPUART_TransferGetSendCountEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle, uint32_t *count)
Gets the number of bytes written to the LPUART TX register.
- status_t [LPUART_TransferGetReceiveCountEDMA](#) (LPUART_Type *base, lpuart_edma_handle_t *handle, uint32_t *count)
Gets the number of received bytes.

24.4.2 Data Structure Documentation

24.4.2.1 struct _lpuart_edma_handle

Data Fields

- `lpuart_edma_transfer_callback_t callback`
Callback function.
- `void *userData`
LPUART callback function parameter.
- `size_t rxDataSizeAll`
Size of the data to receive.
- `size_t txDataSizeAll`
Size of the data to send out.
- `edma_handle_t *txEdmaHandle`
The eDMA TX channel used.
- `edma_handle_t *rxEdmaHandle`
The eDMA RX channel used.
- `uint8_t nbytes`
eDMA minor byte transfer count initially configured.
- `volatile uint8_t txState`
TX transfer state.
- `volatile uint8_t rxState`
RX transfer state.

24.4.2.1.0.68 Field Documentation

24.4.2.1.0.68.1 `lpuart_edma_transfer_callback_t lpuart_edma_handle_t::callback`

24.4.2.1.0.68.2 `void* lpuart_edma_handle_t::userData`

24.4.2.1.0.68.3 `size_t lpuart_edma_handle_t::rxDataSizeAll`

24.4.2.1.0.68.4 `size_t lpuart_edma_handle_t::txDataSizeAll`

24.4.2.1.0.68.5 `edma_handle_t* lpuart_edma_handle_t::txEdmaHandle`

24.4.2.1.0.68.6 `edma_handle_t* lpuart_edma_handle_t::rxEdmaHandle`

24.4.2.1.0.68.7 `uint8_t lpuart_edma_handle_t::nbytes`

24.4.2.1.0.68.8 `volatile uint8_t lpuart_edma_handle_t::txState`

24.4.3 Typedef Documentation

24.4.3.1 `typedef void(* lpuart_edma_transfer_callback_t)(LPUART_Type *base,
lpuart_edma_handle_t *handle, status_t status, void *userData)`

24.4.4 Function Documentation

24.4.4.1 `void LPUART_TransferCreateHandleEDMA (LPUART_Type * base,
lpuart_edma_handle_t * handle, lpuart_edma_transfer_callback_t callback, void
* userData, edma_handle_t * txEdmaHandle, edma_handle_t * rxEdmaHandle)`

LPUART eDMA Driver

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to lpuart_edma_handle_t structure.
<i>callback</i>	Callback function.
<i>userData</i>	User data.
<i>txEdmaHandle</i>	User requested DMA handle for TX DMA transfer.
<i>rxEdmaHandle</i>	User requested DMA handle for RX DMA transfer.

24.4.4.2 status_t LPUART_SendEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*, lpuart_transfer_t * *xfer*)

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

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>xfer</i>	LPUART eDMA transfer structure. See lpuart_transfer_t .

Return values

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

24.4.4.3 status_t LPUART_ReceiveEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*, lpuart_transfer_t * *xfer*)

This function receives data using eDMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to lpuart_edma_handle_t structure.
<i>xfer</i>	LPUART eDMA transfer structure, see lpuart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeed, others fail.
<i>kStatus_LPUART_Rx-Busy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

24.4.4.4 void LPUART_TransferAbortSendEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*)

This function aborts the sent data using eDMA.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to lpuart_edma_handle_t structure.

24.4.4.5 void LPUART_TransferAbortReceiveEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*)

This function aborts the received data using eDMA.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to lpuart_edma_handle_t structure.

24.4.4.6 status_t LPUART_TransferGetSendCountEDMA (LPUART_Type * *base*, lpuart_edma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes written to the LPUART TX register by DMA.

LPUART eDMA Driver

Parameters

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

Return values

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

24.4.4.7 **status_t LPUART_TransferGetReceiveCountEDMA (LPUART_Type * *base*, Ipuart_edma_handle_t * *handle*, uint32_t * *count*)**

This function gets the number of received bytes.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>count</i>	Receive bytes count.

Return values

<i>kStatus_NoTransferIn-Progress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

24.5 LPUART µCOS/II Driver

24.5.1 Overview

Data Structures

- struct **lpuart_rtos_config_t**
LPUART RTOS configuration structure. [More...](#)

LPUART RTOS Operation

- int **LPUART_RTOS_Init** (lpuart_rtos_handle_t *handle, lpuart_handle_t *t_handle, const lpuart_rtos_config_t *cfg)
Initializes an LPUART instance for operation in RTOS.
- int **LPUART_RTOS_Deinit** (lpuart_rtos_handle_t *handle)
Deinitializes an LPUART instance for operation.

LPUART transactional Operation

- int **LPUART_RTOS_Send** (lpuart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
Sends data in the background.
- int **LPUART_RTOS_Receive** (lpuart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
Receives data.

24.5.2 Data Structure Documentation

24.5.2.1 struct lpuart_rtos_config_t

Data Fields

- **LPUART_Type * base**
UART base address.
- **uint32_t srecclk**
UART source clock in Hz.
- **uint32_t baudrate**
Desired communication speed.
- **lpuart_parity_mode_t parity**
Parity setting.
- **lpuart_stop_bit_count_t stopbits**
Number of stop bits to use.
- **uint8_t * buffer**
Buffer for background reception.
- **uint32_t buffer_size**
Size of buffer for background reception.

24.5.3 Function Documentation

24.5.3.1 `int LPUART_RTOS_Init(Ipuart_rtos_handle_t * handle, Ipuart_handle_t * t_handle, const Ipuart_rtos_config_t * cfg)`

Parameters

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

Returns

0 succeed, others failed

24.5.3.2 int LPUART_RTOS_Deinit (*lpuart_rtos_handle_t * handle*)

This function deinitializes the LPUART module, sets all register values to the reset value, and releases the resources.

Parameters

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

24.5.3.3 int LPUART_RTOS_Send (*lpuart_rtos_handle_t * handle, const uint8_t * buffer, uint32_t length*)

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

Parameters

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

24.5.3.4 int LPUART_RTOS_Receive (*lpuart_rtos_handle_t * handle, uint8_t * buffer, uint32_t length, size_t * received*)

This function receives data from LPUART. It is a synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

LPUART µCOS/II Driver

Parameters

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

24.6 LPUART µCOS/III Driver

24.6.1 Overview

Data Structures

- struct **lpuart_rtos_config_t**
LPUART RTOS configuration structure. [More...](#)

LPUART RTOS Operation

- int **LPUART_RTOS_Init** (lpuart_rtos_handle_t *handle, lpuart_handle_t *t_handle, const lpuart_rtos_config_t *cfg)
Initializes an LPUART instance for operation in RTOS.
- int **LPUART_RTOS_Deinit** (lpuart_rtos_handle_t *handle)
Deinitializes an LPUART instance for operation.

LPUART transactional Operation

- int **LPUART_RTOS_Send** (lpuart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
Sends data in the background.
- int **LPUART_RTOS_Receive** (lpuart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
Receives data.

24.6.2 Data Structure Documentation

24.6.2.1 struct lpuart_rtos_config_t

Data Fields

- **LPUART_Type * base**
UART base address.
- **uint32_t srecclk**
UART source clock in Hz.
- **uint32_t baudrate**
Desired communication speed.
- **lpuart_parity_mode_t parity**
Parity setting.
- **lpuart_stop_bit_count_t stopbits**
Number of stop bits to use.
- **uint8_t * buffer**
Buffer for background reception.
- **uint32_t buffer_size**
Size of buffer for background reception.

24.6.3 Function Documentation

24.6.3.1 `int LPUART_RTOS_Init(Ipuart_rtos_handle_t * handle, Ipuart_handle_t * t_handle, const Ipuart_rtos_config_t * cfg)`

Parameters

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

Returns

0 succeed, others failed

24.6.3.2 int LPUART_RTOS_Deinit (*lpuart_rtos_handle_t * handle*)

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

Parameters

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

24.6.3.3 int LPUART_RTOS_Send (*lpuart_rtos_handle_t * handle, const uint8_t * buffer, uint32_t length*)

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

Parameters

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

24.6.3.4 int LPUART_RTOS_Receive (*lpuart_rtos_handle_t * handle, uint8_t * buffer, uint32_t length, size_t * received*)

It is a synchronous API.

This function receives data from LPUART. If any data is immediately available it will be returned imidiately and the number of bytes received.

LPUART µCOS/III Driver

Parameters

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

24.7 LPUART FreeRTOS Driver

24.7.1 Overview

Data Structures

- struct **lpuart_rtos_config_t**
LPUART RTOS configuration structure. [More...](#)

LPUART RTOS Operation

- int **LPUART_RTOS_Init** (lpuart_rtos_handle_t *handle, lpuart_handle_t *t_handle, const **lpuart_rtos_config_t** *cfg)
Initializes an LPUART instance for operation in RTOS.
- int **LPUART_RTOS_Deinit** (lpuart_rtos_handle_t *handle)
Deinitializes an LPUART instance for operation.

LPUART transactional Operation

- int **LPUART_RTOS_Send** (lpuart_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
Sends data in the background.
- int **LPUART_RTOS_Receive** (lpuart_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
Receives data.

24.7.2 Data Structure Documentation

24.7.2.1 struct **lpuart_rtos_config_t**

Data Fields

- **LPUART_Type * base**
UART base address.
- **uint32_t srecclk**
UART source clock in Hz.
- **uint32_t baudrate**
Desired communication speed.
- **lpuart_parity_mode_t parity**
Parity setting.
- **lpuart_stop_bit_count_t stopbits**
Number of stop bits to use.
- **uint8_t * buffer**
Buffer for background reception.
- **uint32_t buffer_size**
Size of buffer for background reception.

24.7.3 Function Documentation

24.7.3.1 `int LPUART_RTOS_Init(Ipuart_rtos_handle_t * handle, Ipuart_handle_t * t_handle, const Ipuart_rtos_config_t * cfg)`

Parameters

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

Returns

0 succeed, others failed

24.7.3.2 int LPUART_RTOS_Deinit (*Ipuart_rtos_handle_t * handle*)

This function deinitializes the LPUART module, sets all register value to the reset value, and releases the resources.

Parameters

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

24.7.3.3 int LPUART_RTOS_Send (*Ipuart_rtos_handle_t * handle, const uint8_t * buffer, uint32_t length*)

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

Parameters

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

24.7.3.4 int LPUART_RTOS_Receive (*Ipuart_rtos_handle_t * handle, uint8_t * buffer, uint32_t length, size_t * received*)

This function receives data from LPUART. It is an synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

LPUART FreeRTOS Driver

Parameters

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

Chapter 25

MPU: Memory Protection Unit

25.1 Overview

The MPU driver provides hardware access control for all memory references generated in the device. Use the MPU driver to program the region descriptors that define memory spaces and their access rights. After initialization, the MPU concurrently monitors the system bus transactions and evaluates their appropriateness.

25.2 Initialization and Deinitialization

To initialize the MPU module, call the [MPU_Init\(\)](#) function and provide the user configuration data structure. This function sets the configuration of the MPU module automatically and enables the MPU module.

Note that the configuration start address, end address, the region valid value, and the debugger's access permission for the MPU region 0 cannot be changed.

This is an example code to configure the MPU driver.

```
// Defines the MPU memory access permission configuration structure. //
mpu_rwxrights_master_access_control_t mpuRwxAccessRightsMasters =
{
    kMPU_SupervisorReadWriteExecute,
    kMPU_UserNoAccessRights,
    kMPU_IdentifierDisable,
    kMPU_SupervisorEqualToUsermode,
    kMPU_UserNoAccessRights,
    kMPU_IdentifierDisable,
    kMPU_SupervisorEqualToUsermode,
    kMPU_UserNoAccessRights,
    kMPU_IdentifierDisable,
    kMPU_SupervisorEqualToUsermode,
    kMPU_UserNoAccessRights,
    kMPU_IdentifierDisable
};

mpu_rwrights_master_access_control_t mpuRwAccessRightsMasters =
{
    false,
    false,
    false,
    false,
    false,
    false,
    false,
    false,
    false
};

// Defines the MPU region configuration structure. //
mpu_region_config_t mpuRegionConfig =
{
    0,
    0x0,
    0xffffffff,
    mpuRwxAccessRightsMasters,
    mpuRwAccessRightsMasters,
```

Basic Control Operations

```
    0,  
    0  
};  
  
// Defines the MPU user configuration structure. //  
mpu_config_t mpuUserConfig =  
{  
    mpuRegionConfig,  
    NULL  
};  
  
// Initializes the MPU region 0. //  
MPU_Init(MPU, &mpuUserConfig);
```

25.3 Basic Control Operations

MPU can be enabled/disabled for the entire memory protection region by calling the [MPU_Enable\(\)](#) function. To save the power for any unused special regions when the entire memory protection region is disabled, call the [MPU_RegionEnable\(\)](#).

After MPU initialization, the [MPU_SetRegionLowMasterAccessRights\(\)](#) and [MPU_SetRegionHighMasterAccessRights\(\)](#) can be used to change the access rights for special master ports and for special region numbers. The [MPU_SetRegionConfig](#) can be used to set the whole region with the start/end address with access rights.

The [MPU_GetHardwareInfo\(\)](#) API is provided to get the hardware information for the device. The [MPU_GetSlavePortErrorStatus\(\)](#) API is provided to get the error status of a special slave port. When an error happens in this port, the [MPU_GetDetailErrorAccessInfo\(\)](#) API is provided to get the detailed error information.

Data Structures

- struct [mpu_hardware_info_t](#)
MPU hardware basic information. [More...](#)
- struct [mpu_access_err_info_t](#)
MPU detail error access information. [More...](#)
- struct [mpu_rwxrights_master_access_control_t](#)
MPU read/write/execute rights control for bus master 0 ~ 3. [More...](#)
- struct [mpu_rwrights_master_access_control_t](#)
MPU read/write access control for bus master 4 ~ 7. [More...](#)
- struct [mpu_region_config_t](#)
MPU region configuration structure. [More...](#)
- struct [mpu_config_t](#)
The configuration structure for the MPU initialization. [More...](#)

Macros

- #define [MPU_REGION_RWXRIGHTS_MASTER_SHIFT\(n\)](#) ($n * 6$)
MPU the bit shift for masters with privilege rights: read write and execute.
- #define [MPU_REGION_RWXRIGHTS_MASTER_MASK\(n\)](#) ($0x1Fu << \text{MPU_REGION_RWXRIGHTS_MASTER_SHIFT}(n)$)
MPU masters with read, write and execute rights bit mask.
- #define [MPU_REGION_RWXRIGHTS_MASTER_WIDTH](#) 5

- `#define MPU_REGION_RWXRIGHTS_MASTER(n, x) (((uint32_t)((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_MASTER_MASK(n)`
MPU masters with read, write and execute rights bit width.
- `#define MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n) (n * 6 + MPU_REGION_RWXRIGHTS_MASTER_WIDTH)`
MPU masters with read, write and execute rights priority setting.
- `#define MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n) (0x1u << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))`
MPU masters with read, write and execute rights process enable bit shift.
- `#define MPU_REGION_RWXRIGHTS_MASTER_PE(n, x) (((uint32_t)((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n)`
MPU masters with read, write and execute rights process enable bit mask.
- `#define MPU_REGION_RWRIGHTS_MASTER_SHIFT(n) ((n - FSL_FEATURE_MPU_PRIVILEGED_RIGHTS_MASTER_COUNT) * 2 + 24)`
MPU masters with read, write and execute rights process enable setting.
- `#define MPU_REGION_RWRIGHTS_MASTER_MASK(n) (0x3u << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))`
MPU masters with normal read write permission bit shift.
- `#define MPU_REGION_RWRIGHTS_MASTER(n, x) (((uint32_t)((uint32_t)(x)) << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWRIGHTS_MASTER_MASTER_MASK(n)`
MPU masters with normal read write rights bit mask.
- `#define MPU_SLAVE_PORT_NUM (4u)`
the Slave port numbers.
- `#define MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX (3)`
define the maximum index of master with privileged rights.

Enumerations

- enum `mpu_region_total_num_t` {

`kMPU_8Regions = 0x0U,`

`kMPU_12Regions = 0x1U,`

`kMPU_16Regions = 0x2U }`

Describes the number of MPU regions.
- enum `mpu_slave_t` {

`kMPU_Slave0 = 0U,`

`kMPU_Slave1 = 1U,`

`kMPU_Slave2 = 2U,`

`kMPU_Slave3 = 3U,`

`kMPU_Slave4 = 4U }`

MPU slave port number.
- enum `mpu_err_access_control_t` {

`kMPU_NoRegionHit = 0U,`

`kMPU_NoneOverlappRegion = 1U,`

`kMPU_OverlappRegion = 2U }`

Basic Control Operations

- *MPU error access control detail.*
 - enum `mpu_err_access_type_t` {
 `kMPU_ErrTypeRead` = 0U,
 `kMPU_ErrTypeWrite` = 1U }
- *MPU error access type.*
 - enum `mpu_err_attributes_t` {
 `kMPU_InstructionAccessInUserMode` = 0U,
 `kMPU_DataAccessInUserMode` = 1U,
 `kMPU_InstructionAccessInSupervisorMode` = 2U,
 `kMPU_DataAccessInSupervisorMode` = 3U }
- *MPU access error attributes.*
 - enum `mpu_supervisor_access_rights_t` {
 `kMPU_SupervisorReadWriteExecute` = 0U,
 `kMPU_SupervisorReadExecute` = 1U,
 `kMPU_SupervisorReadWrite` = 2U,
 `kMPU_SupervisorEqualToUsermode` = 3U }
- *MPU access rights in supervisor mode for bus master 0 ~ 3.*
 - enum `mpu_user_access_rights_t` {
 `kMPU_UserNoAccessRights` = 0U,
 `kMPU_UserExecute` = 1U,
 `kMPU_UserWrite` = 2U,
 `kMPU_UserWriteExecute` = 3U,
 `kMPU_UserRead` = 4U,
 `kMPU_UserReadExecute` = 5U,
 `kMPU_UserReadWrite` = 6U,
 `kMPU_UserReadWriteExecute` = 7U }
- *MPU access rights in user mode for bus master 0 ~ 3.*

Driver version

- #define `FSL_MPUM_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 1)`)
MPU driver version 2.1.0.

Initialization and deinitialization

- void `MPU_Init` (`MPU_Type *base`, const `mpu_config_t *config`)
Initializes the MPU with the user configuration structure.
- void `MPU_Deinit` (`MPU_Type *base`)
Deinitializes the MPU regions.

Basic Control Operations

- static void `MPU_Enable` (`MPU_Type *base`, bool enable)
Enables/disables the MPU globally.
- static void `MPU_RegionEnable` (`MPU_Type *base`, `uint32_t number`, bool enable)
Enables/disables the MPU for a special region.
- void `MPU_GetHardwareInfo` (`MPU_Type *base`, `mpu_hardware_info_t *hardwareInform`)
Gets the MPU basic hardware information.

- void [MPU_SetRegionConfig](#) (MPU_Type *base, const [mpu_region_config_t](#) *regionConfig)
Sets the MPU region.
- void [MPU_SetRegionAddr](#) (MPU_Type *base, uint32_t regionNum, uint32_t startAddr, uint32_t endAddr)
Sets the region start and end address.
- void [MPU_SetRegionRwxMasterAccessRights](#) (MPU_Type *base, uint32_t regionNum, uint32_t masterNum, const [mpu_rwxrights_master_access_control_t](#) *accessRights)
Sets the MPU region access rights for masters with read, write, and execute rights.
- void [MPU_SetRegionRwMasterAccessRights](#) (MPU_Type *base, uint32_t regionNum, uint32_t masterNum, const [mpu_rwrights_master_access_control_t](#) *accessRights)
Sets the MPU region access rights for masters with read and write rights.
- bool [MPU_GetSlavePortErrorStatus](#) (MPU_Type *base, [mpu_slave_t](#) slaveNum)
Gets the numbers of slave ports where errors occur.
- void [MPU_GetDetailErrorAccessInfo](#) (MPU_Type *base, [mpu_slave_t](#) slaveNum, [mpu_access_err_info_t](#) *errInform)
Gets the MPU detailed error access information.

25.4 Data Structure Documentation

25.4.1 struct mpu.hardware_info_t

Data Fields

- uint8_t [hardwareRevisionLevel](#)
Specifies the MPU's hardware and definition reversion level.
- uint8_t [slavePortsNumbers](#)
Specifies the number of slave ports connected to MPU.
- [mpu_region_total_num_t](#) [regionsNumbers](#)
Indicates the number of region descriptors implemented.

25.4.1.0.0.69 Field Documentation

25.4.1.0.0.69.1 uint8_t mpu.hardware_info_t::hardwareRevisionLevel

25.4.1.0.0.69.2 uint8_t mpu.hardware_info_t::slavePortsNumbers

25.4.1.0.0.69.3 [mpu_region_total_num_t](#) mpu.hardware_info_t::regionsNumbers

25.4.2 struct mpu.access_err_info_t

Data Fields

- uint32_t [master](#)
Access error master.
- [mpu_err_attributes_t](#) [attributes](#)
Access error attributes.
- [mpu_err_access_type_t](#) [accessType](#)
Access error type.
- [mpu_err_access_control_t](#) [accessControl](#)

Data Structure Documentation

- `uint32_t address`
Access error address.
- `uint8_t processorIdentification`
Access error processor identification.

25.4.2.0.0.70 Field Documentation

25.4.2.0.0.70.1 `uint32_t mpu_access_err_info_t::master`

25.4.2.0.0.70.2 `mpu_err_attributes_t mpu_access_err_info_t::attributes`

25.4.2.0.0.70.3 `mpu_err_access_type_t mpu_access_err_info_t::accessType`

25.4.2.0.0.70.4 `mpu_err_access_control_t mpu_access_err_info_t::accessControl`

25.4.2.0.0.70.5 `uint32_t mpu_access_err_info_t::address`

25.4.2.0.0.70.6 `uint8_t mpu_access_err_info_t::processorIdentification`

25.4.3 `struct mpu_rwxrights_master_access_control_t`

Data Fields

- `mpu_supervisor_access_rights_t superAccessRights`
Master access rights in supervisor mode.
- `mpu_user_access_rights_t userAccessRights`
Master access rights in user mode.
- `bool processIdentifierEnable`
Enables or disables process identifier.

25.4.3.0.0.71 Field Documentation

25.4.3.0.0.71.1 `mpu_supervisor_access_rights_t mpu_rwxrights_master_access_control_t::superAccessRights`

25.4.3.0.0.71.2 `mpu_user_access_rights_t mpu_rwxrights_master_access_control_t::userAccessRights`

25.4.3.0.0.71.3 `bool mpu_rwxrights_master_access_control_t::processIdentifierEnable`

25.4.4 `struct mpu_rwrights_master_access_control_t`

Data Fields

- `bool writeEnable`
Enables or disables write permission.
- `bool readEnable`
Enables or disables read permission.

25.4.4.0.0.72 Field Documentation**25.4.4.0.0.72.1 bool mpu_rwrights_master_access_control_t::writeEnable****25.4.4.0.0.72.2 bool mpu_rwrights_master_access_control_t::readEnable****25.4.5 struct mpu_region_config_t**

This structure is used to configure the regionNum region. The accessRights1[0] ~ accessRights1[3] are used to configure the bus master 0 ~ 3 with the privilege rights setting. The accessRights2[0] ~ accessRights2[3] are used to configure the high master 4 ~ 7 with the normal read write permission. The master port assignment is the chip configuration. Normally, the core is the master 0, debugger is the master 1. Note that the MPU assigns a priority scheme where the debugger is treated as the highest priority master followed by the core and then all the remaining masters. MPU protection does not allow writes from the core to affect the "regionNum 0" start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters. This protection guarantees that the debugger always has access to the entire address space and those rights can't be changed by the core or any other bus master. Prepare the region configuration when regionNum is 0.

Data Fields

- **uint32_t regionNum**
MPU region number, range form 0 ~ FSL FEATURE_MPU_DESCRIPTOR_COUNT - 1.
- **uint32_t startAddress**
Memory region start address.
- **uint32_t endAddress**
Memory region end address.
- **mpu_rwxrights_master_access_control_t accessRights1 [4]**
Masters with read, write and execute rights setting.
- **mpu_rwrights_master_access_control_t accessRights2 [4]**
Masters with normal read write rights setting.
- **uint8_t processIdentifier**
Process identifier used when "processIdentifierEnable" set with true.
- **uint8_t processIdMask**
Process identifier mask.

25.4.5.0.0.73 Field Documentation**25.4.5.0.0.73.1 uint32_t mpu_region_config_t::regionNum****25.4.5.0.0.73.2 uint32_t mpu_region_config_t::startAddress**

Note: bit0 ~ bit4 always be marked as 0 by MPU. The actual start address is 0-modulo-32 byte address.

25.4.5.0.0.73.3 uint32_t mpu_region_config_t::endAddress

Note: bit0 ~ bit4 always be marked as 1 by MPU. The actual end address is 31-modulo-32 byte address.

Data Structure Documentation

25.4.5.0.0.73.4 `mpu_rwxrights_master_access_control_t mpu_region_config_t::accessRights1[4]`

25.4.5.0.0.73.5 `mpu_rwrights_master_access_control_t mpu_region_config_t::accessRights2[4]`

25.4.5.0.0.73.6 `uint8_t mpu_region_config_t::processIdentifier`

25.4.5.0.0.73.7 `uint8_t mpu_region_config_t::processIdMask`

The setting bit will ignore the same bit in process identifier.

25.4.6 `struct mpu_config_t`

This structure is used when calling the MPU_Init function.

Data Fields

- `mpu_region_config_t regionConfig`
Region access permission.
- `struct _mpu_config * next`
Pointer to the next structure.

25.4.6.0.0.74 Field Documentation

25.4.6.0.0.74.1 `mpu_region_config_t mpu_config_t::regionConfig`

25.4.6.0.0.74.2 `struct _mpu_config* mpu_config_t::next`

25.5 Macro Definition Documentation

- 25.5.1 `#define FSL_MPU_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))`
- 25.5.2 `#define MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n) (n * 6)`
- 25.5.3 `#define MPU_REGION_RWXRIGHTS_MASTER_MASK(n) (0x1Fu << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))`
- 25.5.4 `#define MPU_REGION_RWXRIGHTS_MASTER_WIDTH 5`
- 25.5.5 `#define MPU_REGION_RWXRIGHTS_MASTER(n, x) (((uint32_t)((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_MASK(n))`
- 25.5.6 `#define MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n) (n * 6 + MPU_REGION_RWXRIGHTS_MASTER_WIDTH)`
- 25.5.7 `#define MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n) (0x1u << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))`
- 25.5.8 `#define MPU_REGION_RWXRIGHTS_MASTER_PE(n, x) (((uint32_t)((uint32_t)(x)) << MPU_REGION_RWXRIGHTS_MASTER_PE_SHIFT(n))) & MPU_REGION_RWXRIGHTS_MASTER_PE_MASK(n))`
- 25.5.9 `#define MPU_REGION_RWRIGHTS_MASTER_SHIFT(n) ((n - FSL_FEATURE_MPU_PRIVILEGED_RIGHTS_MASTER_COUNT) * 2 + 24)`
- 25.5.10 `#define MPU_REGION_RWRIGHTS_MASTER_MASK(n) (0x3u << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))`
- 25.5.11 `#define MPU_REGION_RWRIGHTS_MASTER(n, x) (((uint32_t)((uint32_t)(x)) << MPU_REGION_RWRIGHTS_MASTER_SHIFT(n))) & MPU_REGION_RWRIGHTS_MASTER_MASK(n))`
- 25.5.12 `#define MPU_SLAVE_PORT_NUM (4u)`
- 25.5.13 `#define MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX (3)`

Enumeration Type Documentation

25.6 Enumeration Type Documentation

25.6.1 enum mpu_region_total_num_t

Enumerator

kMPU_8Regions MPU supports 8 regions.

kMPU_12Regions MPU supports 12 regions.

kMPU_16Regions MPU supports 16 regions.

25.6.2 enum mpu_slave_t

Enumerator

kMPU_Slave0 MPU slave port 0.

kMPU_Slave1 MPU slave port 1.

kMPU_Slave2 MPU slave port 2.

kMPU_Slave3 MPU slave port 3.

kMPU_Slave4 MPU slave port 4.

25.6.3 enum mpu_err_access_control_t

Enumerator

kMPU_NoRegionHit No region hit error.

kMPU_NoneOverlapRegion Access single region error.

kMPU_OverlapRegion Access overlapping region error.

25.6.4 enum mpu_err_access_type_t

Enumerator

kMPU_ErrTypeRead MPU error access type — read.

kMPU_ErrTypeWrite MPU error access type — write.

25.6.5 enum mpu_err_attributes_t

Enumerator

kMPU_InstructionAccessInUserMode Access instruction error in user mode.

kMPU_DataAccessInUserMode Access data error in user mode.

kMPU_InstructionAccessInSupervisorMode Access instruction error in supervisor mode.

kMPU_DataAccessInSupervisorMode Access data error in supervisor mode.

25.6.6 enum mpu_supervisor_access_rights_t

Enumerator

- kMPU_SupervisorReadWriteExecute*** Read write and execute operations are allowed in supervisor mode.
- kMPU_SupervisorReadExecute*** Read and execute operations are allowed in supervisor mode.
- kMPU_SupervisorReadWrite*** Read write operations are allowed in supervisor mode.
- kMPU_SupervisorEqualToUsermode*** Access permission equal to user mode.

25.6.7 enum mpu_user_access_rights_t

Enumerator

- kMPU_UserNoAccessRights*** No access allowed in user mode.
- kMPU_UserExecute*** Execute operation is allowed in user mode.
- kMPU_UserWrite*** Write operation is allowed in user mode.
- kMPU_UserWriteExecute*** Write and execute operations are allowed in user mode.
- kMPU_UserRead*** Read is allowed in user mode.
- kMPU_UserReadExecute*** Read and execute operations are allowed in user mode.
- kMPU_UserReadWrite*** Read and write operations are allowed in user mode.
- kMPU_UserReadWriteExecute*** Read write and execute operations are allowed in user mode.

25.7 Function Documentation

25.7.1 void MPU_Init (MPU_Type * *base*, const mpu_config_t * *config*)

This function configures the MPU module with the user-defined configuration.

Parameters

<i>base</i>	MPU peripheral base address.
<i>config</i>	The pointer to the configuration structure.

25.7.2 void MPU_Deinit (MPU_Type * *base*)

Parameters

Function Documentation

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

25.7.3 static void MPU_Enable (MPU_Type * *base*, bool *enable*) [inline], [static]

Call this API to enable or disable the MPU module.

Parameters

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

25.7.4 static void MPU_RegionEnable (MPU_Type * *base*, uint32_t *number*, bool *enable*) [inline], [static]

When MPU is enabled, call this API to disable an unused region of an enabled MPU. Call this API to minimize the power dissipation.

Parameters

<i>base</i>	MPU peripheral base address.
<i>number</i>	MPU region number.
<i>enable</i>	True enable the special region MPU, false disable the special region MPU.

25.7.5 void MPU_GetHardwareInfo (MPU_Type * *base*, mpu_hardware_info_t * *hardwareInform*)

Parameters

<i>base</i>	MPU peripheral base address.
<i>hardware-Inform</i>	The pointer to the MPU hardware information structure. See "mpu_hardware_info_t".

25.7.6 void MPU_SetRegionConfig (MPU_Type * *base*, const mpu_region_config_t * *regionConfig*)

Note: Due to the MPU protection, the region number 0 does not allow writes from core to affect the start and end address nor the permissions associated with the debugger. It can only write the permission fields associated with the other masters.

Parameters

<i>base</i>	MPU peripheral base address.
<i>regionConfig</i>	The pointer to the MPU user configuration structure. See "mpu_region_config_t".

25.7.7 void MPU_SetRegionAddr (MPU_Type * *base*, uint32_t *regionNum*, uint32_t *startAddr*, uint32_t *endAddr*)

Memory region start address. Note: bit0 ~ bit4 is always marked as 0 by MPU. The actual start address by MPU is 0-modulo-32 byte address. Memory region end address. Note: bit0 ~ bit4 always be marked as 1 by MPU. The end address used by the MPU is 31-modulo-32 byte address. Note: Due to the MPU protection, the startAddr and endAddr can't be changed by the core when regionNum is 0.

Parameters

<i>base</i>	MPU peripheral base address.
<i>regionNum</i>	MPU region number. The range is from 0 to FSL_FEATURE_MPU_DESCRIPTOR_COUNT - 1.
<i>startAddr</i>	Region start address.
<i>endAddr</i>	Region end address.

25.7.8 void MPU_SetRegionRwxMasterAccessRights (MPU_Type * *base*, uint32_t *regionNum*, uint32_t *masterNum*, const mpu_rwxrights_master_access_control_t * *accessRights*)

The MPU access rights depend on two board classifications of bus masters. The privilege rights masters and the normal rights masters. The privilege rights masters have the read, write, and execute access rights. Except the normal read and write rights, the execute rights are also allowed for these masters. The privilege rights masters normally range from bus masters 0 - 3. However, the maximum master number is device-specific. See the "MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX". The normal rights masters access rights control see "MPU_SetRegionRwMasterAccessRights()".

Function Documentation

Parameters

<i>base</i>	MPU peripheral base address.
<i>regionNum</i>	MPU region number. Should range from 0 to FSL FEATURE MPU_DESCRIPTOR_COUNT - 1.
<i>masterNum</i>	MPU bus master number. Should range from 0 to MPU_PRIVILEGED_RIGHTS_MASTER_MAX_INDEX.
<i>accessRights</i>	The pointer to the MPU access rights configuration. See "mpu_rwxrights_master_access_control_t".

25.7.9 void MPU_SetRegionRwMasterAccessRights (MPU_Type * *base*, uint32_t *regionNum*, uint32_t *masterNum*, const mpu_rwrights_master_access_control_t * *accessRights*)

The MPU access rights depend on two board classifications of bus masters. The privilege rights masters and the normal rights masters. The normal rights masters only have the read and write access permissions. The privilege rights access control see "MPU_SetRegionRwxMasterAccessRights".

Parameters

<i>base</i>	MPU peripheral base address.
<i>regionNum</i>	MPU region number. The range is from 0 to FSL FEATURE MPU_DESCRIPTOR_COUNT - 1.
<i>masterNum</i>	MPU bus master number. Should range from FSL FEATURE MPU_PRIVILEGED_RIGHTS_MASTER_COUNT to ~ FSL FEATURE MPU_MASTER_MAX_INDEX.
<i>accessRights</i>	The pointer to the MPU access rights configuration. See "mpu_rwrights_master_access_control_t".

25.7.10 bool MPU_GetSlavePortErrorStatus (MPU_Type * *base*, mpu_slave_t *slaveNum*)

Parameters

<i>base</i>	MPU peripheral base address.
<i>slaveNum</i>	MPU slave port number.

Returns

The slave ports error status. true - error happens in this slave port. false - error didn't happen in this slave port.

25.7.11 void MPU_GetDetailErrorAccessInfo (**MPU_Type** * *base*, **mpu_slave_t** *slaveNum*, **mpu_access_err_info_t** * *errInform*)

Parameters

<i>base</i>	MPU peripheral base address.
<i>slaveNum</i>	MPU slave port number.
<i>errInform</i>	The pointer to the MPU access error information. See "mpu_access_err_info_t".

Function Documentation

Chapter 26

PDB: Programmable Delay Block

26.1 Overview

The KSDK provides a peripheral driver for the Programmable Delay Block (PDB) module of Kinetis 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 or 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.

26.2 Typical use case

26.2.1 Working as basic PDB counter with a PDB interrupt.

```
int main(void)
{
    // ...
    EnableIRQ(DEMO_PDB IRQ_ID);

    // ...
    // Configures the PDB counter.
    PDB_SetDefaultConfig(& 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);
}
```

26.2.2 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.enableDifferentialConversion = 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_SetDefaultConfig(& 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\_PDDEIE\_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.*

### DAC Interval Trigger

- void [PDB\\_SetDACTriggerConfig](#) (PDB\_Type \*base, uint32\_t channel, [pdb\\_dac\\_trigger\\_config\\_t](#) \*config)  
*Configures the DAC trigger in the PDB module.*
- static void [PDB\\_SetDACTriggerIntervalValue](#) (PDB\_Type \*base, uint32\_t channel, uint32\_t value)  
*Sets the value for the DAC interval event.*

### 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.*

## 26.3 Data Structure Documentation

### 26.3.1 struct [pdb\\_config\\_t](#)

#### Data Fields

- [pdb\\_load\\_value\\_mode\\_t](#) loadValueMode  
*Select the load value mode.*
- [pdb\\_prescaler\\_divider\\_t](#) prescalerDivide  
*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.*

**26.3.1.0.0.75 Field Documentation****26.3.1.0.0.75.1 `adb_load_value_mode_t` `adb_config_t::loadValueMode`****26.3.1.0.0.75.2 `adb_prescaler_divider_t` `adb_config_t::prescalerDivider`****26.3.1.0.0.75.3 `adb_divider_multiplication_factor_t` `adb_config_t::dividerMultiplicationFactor`****26.3.1.0.0.75.4 `adb_trigger_input_source_t` `adb_config_t::triggerInputSource`****26.3.1.0.0.75.5 `bool` `adb_config_t::enableContinuousMode`****26.3.2 `struct adb_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.*

**26.3.2.0.0.76 Field Documentation****26.3.2.0.0.76.1 `uint32_t` `adb_adc_pretrigger_config_t::enablePreTriggerMask`****26.3.2.0.0.76.2 `uint32_t` `adb_adc_pretrigger_config_t::enableOutputMask`**

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

**26.3.2.0.0.76.3 `uint32_t` `adb_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.

**26.3.3 `struct adb_dac_trigger_config_t`****Data Fields**

- `bool enableExternalTriggerInput`  
*Enables the external trigger for DAC interval counter.*
- `bool enableIntervalTrigger`  
*Enables the DAC interval trigger.*

## Enumeration Type Documentation

### 26.3.3.0.0.77 Field Documentation

26.3.3.0.0.77.1 `bool pdb_dac_trigger_config_t::enableExternalTriggerInput`

26.3.3.0.0.77.2 `bool pdb_dac_trigger_config_t::enableIntervalTrigger`

## 26.4 Macro Definition Documentation

26.4.1 `#define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

## 26.5 Enumeration Type Documentation

### 26.5.1 `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.

### 26.5.2 `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.

### 26.5.3 `enum _pdb_interrupt_enable`

Enumerator

*kPDB\_SequenceErrorInterruptEnable* PDB sequence error interrupt enable.

*kPDB\_DelayInterruptEnable* PDB delay interrupt enable.

### 26.5.4 `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.

## 26.5.5 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.

## 26.5.6 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.

## 26.5.7 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.

## Function Documentation

Enumerator

|                             |                                  |
|-----------------------------|----------------------------------|
| <i>kPDB_TriggerInput0</i>   | Trigger-In 0.                    |
| <i>kPDB_TriggerInput1</i>   | Trigger-In 1.                    |
| <i>kPDB_TriggerInput2</i>   | Trigger-In 2.                    |
| <i>kPDB_TriggerInput3</i>   | Trigger-In 3.                    |
| <i>kPDB_TriggerInput4</i>   | Trigger-In 4.                    |
| <i>kPDB_TriggerInput5</i>   | Trigger-In 5.                    |
| <i>kPDB_TriggerInput6</i>   | Trigger-In 6.                    |
| <i>kPDB_TriggerInput7</i>   | Trigger-In 7.                    |
| <i>kPDB_TriggerInput8</i>   | Trigger-In 8.                    |
| <i>kPDB_TriggerInput9</i>   | Trigger-In 9.                    |
| <i>kPDB_TriggerInput10</i>  | Trigger-In 10.                   |
| <i>kPDB_TriggerInput11</i>  | Trigger-In 11.                   |
| <i>kPDB_TriggerInput12</i>  | Trigger-In 12.                   |
| <i>kPDB_TriggerInput13</i>  | Trigger-In 13.                   |
| <i>kPDB_TriggerInput14</i>  | Trigger-In 14.                   |
| <i>kPDB_TriggerSoftware</i> | Trigger-In 15, software trigger. |

## 26.6 Function Documentation

### 26.6.1 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". |

### 26.6.2 void PDB\_Deinit ( **PDB\_Type** \* *base* )

Parameters

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

### 26.6.3 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". |
|---------------|---------------------------------------------------------|

### 26.6.4 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.    |

### 26.6.5 static void PDB\_DoSoftwareTrigger ( *PDB\_Type \* base* ) [inline], [static]

Parameters

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

### 26.6.6 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 PD-B's load mode.

## Function Documentation

Parameters

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

**26.6.7 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.   |

**26.6.8 static void PDB\_EnableInterrupts ( 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". |

**26.6.9 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". |

**26.6.10 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".

#### 26.6.11 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". |

#### 26.6.12 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. |

#### 26.6.13 static uint32\_t PDB\_GetCounterValue ( PDB\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

PDB counter's current value.

#### 26.6.14 static void PDB\_SetCounterDelayValue ( PDB\_Type \* *base*, uint32\_t *value* ) [inline], [static]

## Function Documentation

Parameters

|              |                                                                 |
|--------------|-----------------------------------------------------------------|
| <i>base</i>  | PDB peripheral base address.                                    |
| <i>value</i> | Setting value for PDB counter delay event. 16-bit is available. |

**26.6.15 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". |

**26.6.16 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. |

**26.6.17 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".

#### 26.6.18 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". |

#### 26.6.19 void PDB\_SetDACTriggerConfig ( PDB\_Type \* *base*, uint32\_t *channel*, pdb\_dac\_trigger\_config\_t \* *config* )

Parameters

|                |                                                                         |
|----------------|-------------------------------------------------------------------------|
| <i>base</i>    | PDB peripheral base address.                                            |
| <i>channel</i> | Channel index for DAC instance.                                         |
| <i>config</i>  | Pointer to the configuration structure. See "pdb_dac_trigger_config_t". |

#### 26.6.20 static void PDB\_SetDACTriggerIntervalValue ( PDB\_Type \* *base*, uint32\_t *channel*, uint32\_t *value* ) [inline], [static]

This function sets the value for DAC interval event. DAC interval trigger triggers the DAC module to update the buffer when the DAC interval counter is equal to the set value.

Parameters

## Function Documentation

|                |                                           |
|----------------|-------------------------------------------|
| <i>base</i>    | PDB peripheral base address.              |
| <i>channel</i> | Channel index for DAC instance.           |
| <i>value</i>   | Setting value for the DAC interval event. |

**26.6.21 static void PDB\_EnablePulseOutTrigger ( PDB\_Type \* *base*, uint32\_t *channelMask*, bool *enable* ) [inline], [static]**

Parameters

|                    |                                                            |
|--------------------|------------------------------------------------------------|
| <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.                     |

**26.6.22 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 27

## PIT: Periodic Interrupt Timer

### 27.1 Overview

The KSDK provides a driver for the Periodic Interrupt Timer (PIT) of Kinetis devices.

### 27.2 Function groups

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

#### 27.2.1 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.

#### 27.2.2 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.

#### 27.2.3 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

### 27.2.4 Status

Provides functions to get and clear the PIT status.

### 27.2.5 Interrupt

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

## 27.3 Typical use case

### 27.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically.

```
int main(void)
{
 /* Structure of initialize PIT */
 pit_config_t pitConfig;

 /* Initialize and enable LED */
 LED_INIT();

 /* Board pin, clock, debug console init */
 BOARD_InitHardware();

 PIT_GetDefaultConfig(&pitConfig);

 /* Init pit module */
 PIT_Init(PIT, &pitConfig);

 /* Set timer period for channel 0 */
 PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000000U,
 PIT_SOURCE_CLOCK));

 /* Enable timer interrupts for channel 0 */
 PIT_EnableInterrupts(PIT, kPIT_Chnl_0,
 kPIT_TimerInterruptEnable);

 /* Enable at the NVIC */
 EnableIRQ(PIT IRQ_ID);

 /* Start channel 0 */
 PRINTF("\r\nStarting channel No.0 ...");
 PIT_StartTimer(PIT, kPIT_Chnl_0);

 while (true)
 {
 /* Check whether occur interrupt and toggle LED */
 if (true == pitIsrFlag)
 {
 PRINTF("\r\n Channel No.0 interrupt is occurred !");
 LED_TOGGLE();
 pitIsrFlag = false;
 }
 }
}
```

## Data Structures

- struct `pit_config_t`  
*PIT configuration structure.* [More...](#)

## Enumerations

- enum `pit_chnl_t` {
   
`kPIT_Chnl_0` = 0U,  
`kPIT_Chnl_1`,  
`kPIT_Chnl_2`,  
`kPIT_Chnl_3` }
   
*List of PIT channels.*
- enum `pit_interrupt_enable_t` { `kPIT_TimerInterruptEnable` = PIT\_TCTRL\_TIE\_MASK }  
*List of PIT interrupts.*
- enum `pit_status_flags_t` { `kPIT_TimerFlag` = PIT\_TFLG\_TIF\_MASK }  
*List of PIT status flags.*

## Functions

- `uint64_t PIT_GetLifetimeTimerCount (PIT_Type *base)`  
*Reads the current lifetime counter value.*

## Driver version

- `#define FSL_PIT_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`  
*Version 2.0.0.*

## Initialization and deinitialization

- `void PIT_Init (PIT_Type *base, const pit_config_t *config)`  
*Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.*
- `void PIT_Deinit (PIT_Type *base)`  
*Gates the PIT clock and disables the PIT module.*
- `static void PIT_GetDefaultConfig (pit_config_t *config)`  
*Fills in the PIT configuration structure with the default settings.*
- `static void PIT_SetTimerChainMode (PIT_Type *base, pit_chnl_t channel, bool enable)`  
*Enables or disables chaining a timer with the previous timer.*

## Interrupt Interface

- `static void PIT_EnableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)`  
*Enables the selected PIT interrupts.*
- `static void PIT_DisableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)`  
*Disables the selected PIT interrupts.*
- `static uint32_t PIT_GetEnabledInterrupts (PIT_Type *base, pit_chnl_t channel)`  
*Gets the enabled PIT interrupts.*

## Enumeration Type Documentation

### Status Interface

- static uint32\_t [PIT\\_GetStatusFlags](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Gets the PIT status flags.*
- static void [PIT\\_ClearStatusFlags](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel, uint32\_t mask)  
*Clears the PIT status flags.*

### Read and Write the timer period

- static void [PIT\\_SetTimerPeriod](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel, uint32\_t count)  
*Sets the timer period in units of count.*
- static uint32\_t [PIT\\_GetCurrentTimerCount](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Reads the current timer counting value.*

### Timer Start and Stop

- static void [PIT\\_StartTimer](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Starts the timer counting.*
- static void [PIT\\_StopTimer](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Stops the timer counting.*

## 27.4 Data Structure Documentation

### 27.4.1 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*

## 27.5 Enumeration Type Documentation

### 27.5.1 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.

### 27.5.2 enum pit\_interrupt\_enable\_t

Enumerator

***kPIT\_TimerInterruptEnable*** Timer interrupt enable.

### 27.5.3 enum pit\_status\_flags\_t

Enumerator

***kPIT\_TimerFlag*** Timer flag.

## 27.6 Function Documentation

### 27.6.1 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 |

### 27.6.2 void PIT\_Deinit ( PIT\_Type \* *base* )

Parameters

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

### 27.6.3 static void PIT\_GetDefaultConfig ( pit\_config\_t \* *config* ) [inline], [static]

The default values are as follows.

```
* config->enableRunInDebug = false;
*
```

## Function Documentation

Parameters

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

### 27.6.4 static void PIT\_SetTimerChainMode ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **bool** *enable* ) [inline], [static]

When a timer has a chain mode enabled, it only counts after the previous timer has expired. If the timer n-1 has counted down to 0, counter n decrements the value by one. Each timer is 32-bits, which allows the developers to chain timers together and form a longer timer (64-bits and larger). The first timer (timer 0) can't be chained to any other timer.

Parameters

|                |                                                                                                                                |
|----------------|--------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                                    |
| <i>channel</i> | Timer channel number which is chained with the previous timer                                                                  |
| <i>enable</i>  | Enable or disable chain. true: Current timer is chained with the previous timer. false: Timer doesn't chain with other timers. |

### 27.6.5 static void PIT\_EnableInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|                |                                                                                                                     |
|----------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                         |
| <i>channel</i> | Timer channel number                                                                                                |
| <i>mask</i>    | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a> |

### 27.6.6 static void PIT\_DisableInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|                |                                                                                                                      |
|----------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                          |
| <i>channel</i> | Timer channel number                                                                                                 |
| <i>mask</i>    | The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a> |

### 27.6.7 static uint32\_t PIT\_GetEnabledInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [inline], [static]

Parameters

|                |                             |
|----------------|-----------------------------|
| <i>base</i>    | PIT peripheral base address |
| <i>channel</i> | Timer channel number        |

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [pit\\_interrupt\\_enable\\_t](#)

### 27.6.8 static uint32\_t PIT\_GetStatusFlags ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [inline], [static]

Parameters

|                |                             |
|----------------|-----------------------------|
| <i>base</i>    | PIT peripheral base address |
| <i>channel</i> | Timer channel number        |

Returns

The status flags. This is the logical OR of members of the enumeration [pit\\_status\\_flags\\_t](#)

### 27.6.9 static void PIT\_ClearStatusFlags ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

## Function Documentation

Parameters

|                |                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                      |
| <i>channel</i> | Timer channel number                                                                                             |
| <i>mask</i>    | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">pit_status_flags_t</a> |

### 27.6.10 static void PIT\_SetTimerPeriod( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *count* ) [inline], [static]

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note

Users can call the utility macros provided in fsl\_common.h to convert to ticks.

Parameters

|                |                                |
|----------------|--------------------------------|
| <i>base</i>    | PIT peripheral base address    |
| <i>channel</i> | Timer channel number           |
| <i>count</i>   | Timer period in units of ticks |

### 27.6.11 static uint32\_t PIT\_GetCurrentTimerCount( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [inline], [static]

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl\_common.h to convert ticks to usec or msec.

Parameters

|                |                             |
|----------------|-----------------------------|
| <i>base</i>    | PIT peripheral base address |
| <i>channel</i> | Timer channel number        |

Returns

Current timer counting value in ticks

#### 27.6.12 static void PIT\_StartTimer ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [**inline**], [**static**]

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

|                |                             |
|----------------|-----------------------------|
| <i>base</i>    | PIT peripheral base address |
| <i>channel</i> | Timer channel number.       |

#### 27.6.13 static void PIT\_StopTimer ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [**inline**], [**static**]

This function stops every timer counting. Timers reload their periods respectively after the next time they call the PIT\_DRV\_StartTimer.

Parameters

|                |                             |
|----------------|-----------------------------|
| <i>base</i>    | PIT peripheral base address |
| <i>channel</i> | Timer channel number.       |

#### 27.6.14 uint64\_t PIT\_GetLifetimeTimerCount ( **PIT\_Type** \* *base* )

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

## Function Documentation

Parameters

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

Returns

Current lifetime timer value

# Chapter 28

## PMC: Power Management Controller

### 28.1 Overview

The KSDK provides a Peripheral driver for the Power Management Controller (PMC) module of Kinetis devices. The PMC module contains internal voltage regulator, power on reset, low-voltage detect system, and high-voltage detect system.

### Data Structures

- struct [pmc\\_low\\_volt\\_detect\\_config\\_t](#)  
*Low-voltage Detect Configuration Structure. [More...](#)*
- struct [pmc\\_low\\_volt\\_warning\\_config\\_t](#)  
*Low-voltage Warning Configuration Structure. [More...](#)*
- struct [pmc\\_bandgap\\_buffer\\_config\\_t](#)  
*Bandgap Buffer configuration. [More...](#)*

### Enumerations

- enum [pmc\\_low\\_volt\\_detect\\_volt\\_select\\_t](#) {  
  kPMC\_LowVoltDetectLowTrip = 0U,  
  kPMC\_LowVoltDetectHighTrip = 1U }  
*Low-voltage Detect Voltage Select.*
- enum [pmc\\_low\\_volt\\_warning\\_volt\\_select\\_t](#) {  
  kPMC\_LowVoltWarningLowTrip = 0U,  
  kPMC\_LowVoltWarningMid1Trip = 1U,  
  kPMC\_LowVoltWarningMid2Trip = 2U,  
  kPMC\_LowVoltWarningHighTrip = 3U }  
*Low-voltage Warning Voltage Select.*

### Driver version

- #define [FSL\\_PMC\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 0))  
*PMC driver version.*

### Power Management Controller Control APIs

- void [PMC\\_ConfigureLowVoltDetect](#) (PMC\_Type \*base, const [pmc\\_low\\_volt\\_detect\\_config\\_t](#) \*config)  
*Configures the low-voltage detect setting.*
- static bool [PMC\\_GetLowVoltDetectFlag](#) (PMC\_Type \*base)  
*Gets the Low-voltage Detect Flag status.*
- static void [PMC\\_ClearLowVoltDetectFlag](#) (PMC\_Type \*base)  
*Acknowledges clearing the Low-voltage Detect flag.*

## Data Structure Documentation

- void [PMC\\_ConfigureLowVoltWarning](#) (PMC\_Type \*base, const [pmc\\_low\\_volt\\_warning\\_config\\_t](#) \*config)  
*Configures the low-voltage warning setting.*
- static bool [PMC\\_GetLowVoltWarningFlag](#) (PMC\_Type \*base)  
*Gets the Low-voltage Warning Flag status.*
- static void [PMC\\_ClearLowVoltWarningFlag](#) (PMC\_Type \*base)  
*Acknowledges the Low-voltage Warning flag.*
- void [PMC\\_ConfigureBandgapBuffer](#) (PMC\_Type \*base, const [pmc\\_bandgap\\_buffer\\_config\\_t](#) \*config)  
*Configures the PMC bandgap.*
- static bool [PMC\\_GetPeriphIOIsolationFlag](#) (PMC\_Type \*base)  
*Gets the acknowledge Peripherals and I/O pads isolation flag.*
- static void [PMC\\_ClearPeriphIOIsolationFlag](#) (PMC\_Type \*base)  
*Acknowledges the isolation flag to Peripherals and I/O pads.*
- static bool [PMC\\_IsRegulatorInRunRegulation](#) (PMC\_Type \*base)  
*Gets the regulator regulation status.*

## 28.2 Data Structure Documentation

### 28.2.1 struct pmc\_low\_volt\_detect\_config\_t

#### Data Fields

- bool [enableInt](#)  
*Enable interrupt when Low-voltage detect.*
- bool [enableReset](#)  
*Enable system reset when Low-voltage detect.*
- [pmc\\_low\\_volt\\_detect\\_volt\\_select\\_t](#) [voltSelect](#)  
*Low-voltage detect trip point voltage selection.*

### 28.2.2 struct pmc\_low\_volt\_warning\_config\_t

#### Data Fields

- bool [enableInt](#)  
*Enable interrupt when low-voltage warning.*
- [pmc\\_low\\_volt\\_warning\\_volt\\_select\\_t](#) [voltSelect](#)  
*Low-voltage warning trip point voltage selection.*

### 28.2.3 struct pmc\_bandgap\_buffer\_config\_t

#### Data Fields

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

*Enable bandgap buffer in low-power mode.*

### 28.2.3.0.0.78 Field Documentation

28.2.3.0.0.78.1 `bool pmc_bandgap_buffer_config_t::enable`

28.2.3.0.0.78.2 `bool pmc_bandgap_buffer_config_t::enableInLowPowerMode`

## 28.3 Macro Definition Documentation

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

Version 2.0.0.

## 28.4 Enumeration Type Documentation

28.4.1 `enum pmc_low_volt_detect_volt_select_t`

Enumerator

*kPMC\_LowVoltDetectLowTrip* Low-trip point selected (VLVD = VLVDL )

*kPMC\_LowVoltDetectHighTrip* High-trip point selected (VLVD = VLVDH )

28.4.2 `enum pmc_low_volt_warning_volt_select_t`

Enumerator

*kPMC\_LowVoltWarningLowTrip* Low-trip point selected (VLVW = VLVW1)

*kPMC\_LowVoltWarningMid1Trip* Mid 1 trip point selected (VLVW = VLVW2)

*kPMC\_LowVoltWarningMid2Trip* Mid 2 trip point selected (VLVW = VLVW3)

*kPMC\_LowVoltWarningHighTrip* High-trip point selected (VLVW = VLVW4)

## 28.5 Function Documentation

28.5.1 `void PMC_ConfigureLowVoltDetect ( PMC_Type * base, const pmc_low_volt_detect_config_t * config )`

This function configures the low-voltage detect setting, including the trip point voltage setting, enables or disables the interrupt, enables or disables the system reset.

Parameters

## Function Documentation

|               |                                             |
|---------------|---------------------------------------------|
| <i>base</i>   | PMC peripheral base address.                |
| <i>config</i> | Low-voltage detect configuration structure. |

### 28.5.2 static bool PMC\_GetLowVoltDetectFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function reads the current LVDF status. If it returns 1, a low-voltage event is detected.

Parameters

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

Returns

- Current low-voltage detect flag
- true: Low-voltage detected
  - false: Low-voltage not detected

### 28.5.3 static void PMC\_ClearLowVoltDetectFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function acknowledges the low-voltage detection errors (write 1 to clear LVDF).

Parameters

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

### 28.5.4 void PMC\_ConfigureLowVoltWarning ( **PMC\_Type** \* *base*, const *pmc\_low\_volt\_warning\_config\_t* \* *config* )

This function configures the low-voltage warning setting, including the trip point voltage setting and enabling or disabling the interrupt.

Parameters

|               |                                              |
|---------------|----------------------------------------------|
| <i>base</i>   | PMC peripheral base address.                 |
| <i>config</i> | Low-voltage warning configuration structure. |

### 28.5.5 static bool PMC\_GetLowVoltWarningFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

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

Returns

Current LVWF status

- true: Low-voltage Warning Flag is set.
- false: the Low-voltage Warning does not happen.

### 28.5.6 static void PMC\_ClearLowVoltWarningFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

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

### 28.5.7 void PMC\_ConfigureBandgapBuffer ( **PMC\_Type** \* *base*, const **pmc\_bandgap\_buffer\_config\_t** \* *config* )

This function configures the PMC bandgap, including the drive select and behavior in low-power mode.

Parameters

## Function Documentation

|               |                                        |
|---------------|----------------------------------------|
| <i>base</i>   | PMC peripheral base address.           |
| <i>config</i> | Pointer to the configuration structure |

### 28.5.8 static bool PMC\_GetPeriphIOIsolationFlag ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function reads the Acknowledge Isolation setting that indicates whether certain peripherals and the I/O pads are in a latched state as a result of having been in the VLLS mode.

Parameters

|             |                                        |
|-------------|----------------------------------------|
| <i>base</i> | PMC peripheral base address.           |
| <i>base</i> | Base address for current PMC instance. |

Returns

ACK isolation 0 - Peripherals and I/O pads are in a normal run state. 1 - Certain peripherals and I/O pads are in an isolated and latched state.

### 28.5.9 static void PMC\_ClearPeriphIOIsolationFlag ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function clears the ACK Isolation flag. Writing one to this setting when it is set releases the I/O pads and certain peripherals to their normal run mode state.

Parameters

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

### 28.5.10 static bool PMC\_IsRegulatorInRunRegulation ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function returns the regulator to run a regulation status. It provides the current status of the internal voltage regulator.

## Parameters

|             |                                        |
|-------------|----------------------------------------|
| <i>base</i> | PMC peripheral base address.           |
| <i>base</i> | Base address for current PMC instance. |

## Returns

Regulation status 0 - Regulator is in a stop regulation or in transition to/from the regulation. 1 - Regulator is in a run regulation.

## Function Documentation

# Chapter 29

## PORT: Port Control and Interrupts

### 29.1 Overview

The KSDK provides a driver for the Port Control and Interrupts (PORT) module of Kinetis devices.

### 29.2 Typical configuration use case

#### 29.2.1 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);
```

#### 29.2.2 I2C PORT Configuration

```
/* I2C pin PORTconfiguration */
port_pin_config_t config = {
 kPORT_PullUp,
 kPORT_FastSlewRate,
 kPORT_PassiveFilterDisable,
 kPORT_OpenDrainEnable,
 kPORT_LowDriveStrength,
 kPORT_MuxAlt5,
 kPORT_UnLockRegister,
};

PORT_SetPinConfig(PORTE, 24u, &config);
PORT_SetPinConfig(PORTE, 25u, &config);
```

## Data Structures

- struct [port\\_digital\\_filter\\_config\\_t](#)  
*PORT digital filter feature configuration definition.* [More...](#)
- struct [port\\_pin\\_config\\_t](#)  
*PORT pin configuration structure.* [More...](#)

## Enumerations

- enum [\\_port\\_pull](#) {  
    kPORT\_PullDisable = 0U,  
    kPORT\_PullDown = 2U,

## Typical configuration use case

- ```
kPORT_PullUp = 3U }  
    Internal resistor pull feature selection.  
• enum _port_slew_rate {  
    kPORT_FastSlewRate = 0U,  
    kPORT_SlowSlewRate = 1U }  
        Slew rate selection.  
• enum _port_open_drain_enable {  
    kPORT_OpenDrainDisable = 0U,  
    kPORT_OpenDrainEnable = 1U }  
        Open Drain feature enable/disable.  
• enum _port_passive_filter_enable {  
    kPORT_PassiveFilterDisable = 0U,  
    kPORT_PassiveFilterEnable = 1U }  
        Passive filter feature enable/disable.  
• enum _port_drive_strength {  
    kPORT_LowDriveStrength = 0U,  
    kPORT_HighDriveStrength = 1U }  
        Configures the drive strength.  
• enum _port_lock_register {  
    kPORT_UnlockRegister = 0U,  
    kPORT_LockRegister = 1U }  
        Unlock/lock the pin control register field[15:0].  
• enum port_mux_t {  
    kPORT_PinDisabledOrAnalog = 0U,  
    kPORT_MuxAsGpio = 1U,  
    kPORT_MuxAlt2 = 2U,  
    kPORT_MuxAlt3 = 3U,  
    kPORT_MuxAlt4 = 4U,  
    kPORT_MuxAlt5 = 5U,  
    kPORT_MuxAlt6 = 6U,  
    kPORT_MuxAlt7 = 7U,  
    kPORT_MuxAlt8 = 8U,  
    kPORT_MuxAlt9 = 9U,  
    kPORT_MuxAlt10 = 10U,  
    kPORT_MuxAlt11 = 11U,  
    kPORT_MuxAlt12 = 12U,  
    kPORT_MuxAlt13 = 13U,  
    kPORT_MuxAlt14 = 14U,  
    kPORT_MuxAlt15 = 15U }  
        Pin mux selection.  
• enum port_interrupt_t {
```

```
kPORT_InterruptOrDMADisabled = 0x0U,
kPORT_DMARisingEdge = 0x1U,
kPORT_DMAFallingEdge = 0x2U,
kPORT_DMAEitherEdge = 0x3U,
kPORT_InterruptLogicZero = 0x8U,
kPORT_InterruptRisingEdge = 0x9U,
kPORT_InterruptFallingEdge = 0xAU,
kPORT_InterruptEitherEdge = 0xBU,
kPORT_InterruptLogicOne = 0xCU }
```

Configures the interrupt generation condition.

- enum **port_digital_filter_clock_source_t** {
 kPORT_BusClock = 0U,
 kPORT_LpoClock = 1U }

Digital filter clock source selection.

Driver version

- #define **FSL_PORT_DRIVER_VERSION** (MAKE_VERSION(2, 0, 2))
Version 2.0.2.

Configuration

- static void **PORT_SetPinConfig** (PORT_Type *base, uint32_t pin, const **port_pin_config_t** *config)
Sets the port PCR register.
- static void **PORT_SetMultiplePinsConfig** (PORT_Type *base, uint32_t mask, const **port_pin_config_t** *config)
Sets the port PCR register for multiple pins.
- static void **PORT_SetPinMux** (PORT_Type *base, uint32_t pin, **port_mux_t** mux)
Configures the pin muxing.
- static void **PORT_EnablePinsDigitalFilter** (PORT_Type *base, uint32_t mask, bool enable)
Enables the digital filter in one port, each bit of the 32-bit register represents one pin.
- static void **PORT_SetDigitalFilterConfig** (PORT_Type *base, const **port_digital_filter_config_t** *config)
Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Interrupt

- static void **PORT_SetPinInterruptConfig** (PORT_Type *base, uint32_t pin, **port_interrupt_t** config)
Configures the port pin interrupt/DMA request.
- static uint32_t **PORT_GetPinsInterruptFlags** (PORT_Type *base)
Reads the whole port status flag.
- static void **PORT_ClearPinsInterruptFlags** (PORT_Type *base, uint32_t mask)
Clears the multiple pin interrupt status flag.

Enumeration Type Documentation

29.3 Data Structure Documentation

29.3.1 struct port_digital_filter_config_t

Data Fields

- `uint32_t digitalFilterWidth`
Set digital filter width.
- `port_digital_filter_clock_source_t clockSource`
Set digital filter clockSource.

29.3.2 struct port_pin_config_t

Data Fields

- `uint16_t pullSelect: 2`
No-pull/pull-down/pull-up select.
- `uint16_t slewRate: 1`
Fast/slow slew rate Configure.
- `uint16_t passiveFilterEnable: 1`
Passive filter enable/disable.
- `uint16_t openDrainEnable: 1`
Open drain enable/disable.
- `uint16_t driveStrength: 1`
Fast/slow drive strength configure.
- `uint16_t mux: 3`
Pin mux Configure.
- `uint16_t lockRegister: 1`
Lock/unlock the PCR field[15:0].

29.4 Macro Definition Documentation

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

29.5 Enumeration Type Documentation

29.5.1 enum _port_pull

Enumerator

kPORT_PullDisable Internal pull-up/down resistor is disabled.

kPORT_PullDown Internal pull-down resistor is enabled.

kPORT_PullUp Internal pull-up resistor is enabled.

29.5.2 enum _port_slew_rate

Enumerator

kPORT_FastSlewRate Fast slew rate is configured.

kPORT_SlowSlewRate Slow slew rate is configured.

29.5.3 enum _port_open_drain_enable

Enumerator

kPORT_OpenDrainDisable Open drain output is disabled.

kPORT_OpenDrainEnable Open drain output is enabled.

29.5.4 enum _port_passive_filter_enable

Enumerator

kPORT_PassiveFilterDisable Passive input filter is disabled.

kPORT_PassiveFilterEnable Passive input filter is enabled.

29.5.5 enum _port_drive_strength

Enumerator

kPORT_LowDriveStrength Low-drive strength is configured.

kPORT_HighDriveStrength High-drive strength is configured.

29.5.6 enum _port_lock_register

Enumerator

kPORT_UnlockRegister Pin Control Register fields [15:0] are not locked.

kPORT_LockRegister Pin Control Register fields [15:0] are locked.

29.5.7 enum port_mux_t

Enumerator

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

Function Documentation

kPORT_MuxAsGpio Corresponding pin is configured as GPIO.
kPORT_MuxAlt2 Chip-specific.
kPORT_MuxAlt3 Chip-specific.
kPORT_MuxAlt4 Chip-specific.
kPORT_MuxAlt5 Chip-specific.
kPORT_MuxAlt6 Chip-specific.
kPORT_MuxAlt7 Chip-specific.
kPORT_MuxAlt8 Chip-specific.
kPORT_MuxAlt9 Chip-specific.
kPORT_MuxAlt10 Chip-specific.
kPORT_MuxAlt11 Chip-specific.
kPORT_MuxAlt12 Chip-specific.
kPORT_MuxAlt13 Chip-specific.
kPORT_MuxAlt14 Chip-specific.
kPORT_MuxAlt15 Chip-specific.

29.5.8 enum port_interrupt_t

Enumerator

kPORT_InterruptOrDMADisabled Interrupt/DMA request is disabled.
kPORT_DMARisingEdge DMA request on rising edge.
kPORT_DMAFallingEdge DMA request on falling edge.
kPORT_DMAEitherEdge DMA request on either edge.
kPORT_InterruptLogicZero Interrupt when logic zero.
kPORT_InterruptRisingEdge Interrupt on rising edge.
kPORT_InterruptFallingEdge Interrupt on falling edge.
kPORT_InterruptEitherEdge Interrupt on either edge.
kPORT_InterruptLogicOne Interrupt when logic one.

29.5.9 enum port_digital_filter_clock_source_t

Enumerator

kPORT_BusClock Digital filters are clocked by the bus clock.
kPORT_LpoClock Digital filters are clocked by the 1 kHz LPO clock.

29.6 Function Documentation

29.6.1 static void PORT_SetPinConfig (PORT_Type * *base*, uint32_t *pin*, const port_pin_config_t * *config*) [inline], [static]

This is an example to define an input pin or output pin PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
*     kPORT_PullUp,
*     kPORT_FastSlewRate,
*     kPORT_PassiveFilterDisable,
*     kPORT_OpenDrainDisable,
*     kPORT_LowDriveStrength,
*     kPORT_MuxAsGpio,
*     kPORT_UnLockRegister,
* };
*
```

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>config</i>	PORT PCR register configuration structure.

29.6.2 static void PORT_SetMultiplePinsConfig (PORT_Type * *base*, uint32_t *mask*, const port_pin_config_t * *config*) [inline], [static]

This is an example to define input pins or output pins PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
*     kPORT_PullUp ,
*     kPORT_PullEnable,
*     kPORT_FastSlewRate,
*     kPORT_PassiveFilterDisable,
*     kPORT_OpenDrainDisable,
*     kPORT_LowDriveStrength,
*     kPORT_MuxAsGpio,
*     kPORT_UnlockRegister,
* };
*
```

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.
<i>config</i>	PORT PCR register configuration structure.

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

Function Documentation

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>mux</i>	pin muxing slot selection. <ul style="list-style-type: none">• kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function.• kPORT_MuxAsGpio : Set as GPIO.• kPORT_MuxAlt2 : chip-specific.• kPORT_MuxAlt3 : chip-specific.• kPORT_MuxAlt4 : chip-specific.• kPORT_MuxAlt5 : chip-specific.• kPORT_MuxAlt6 : chip-specific.• kPORT_MuxAlt7 : chip-specific. : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

29.6.4 static void PORT_EnablePinsDigitalFilter (PORT_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.

29.6.5 static void PORT_SetDigitalFilterConfig (PORT_Type * *base*, const port_digital_filter_config_t * *config*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>config</i>	PORT digital filter configuration structure.

29.6.6 static void PORT_SetPinInterruptConfig (PORT_Type * *base*, uint32_t *pin*, port_interrupt_t *config*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>config</i>	<p>PORT pin interrupt configuration.</p> <ul style="list-style-type: none"> • kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled. • kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit). • kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit). • kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit). • #kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit). • #kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit). • #kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit). • kPORT_InterruptLogicZero : Interrupt when logic zero. • kPORT_InterruptRisingEdge : Interrupt on rising edge. • kPORT_InterruptFallingEdge: Interrupt on falling edge. • kPORT_InterruptEitherEdge : Interrupt on either edge. • kPORT_InterruptLogicOne : Interrupt when logic one. • #kPORT_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit). • #kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

29.6.7 static uint32_t PORT_GetPinsInterruptFlags (PORT_Type * *base*) [inline], [static]

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

<i>base</i>	PORT peripheral base pointer.
-------------	-------------------------------

Returns

Current port interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

Function Documentation

29.6.8 static void PORT_ClearPinsInterruptFlags (PORT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.

Function Documentation

Chapter 30

RCM: Reset Control Module Driver

30.1 Overview

The KSDK provides a Peripheral driver for the Reset Control Module (RCM) module of Kinetis devices.

Data Structures

- struct `rcm_reset_pin_filter_config_t`
Reset pin filter configuration. [More...](#)

Enumerations

- enum `rcm_reset_source_t` {
 `kRCM_SourceWakeup` = RCM_SRS0_WAKEUP_MASK,
 `kRCM_SourceLvd` = RCM_SRS0_LVD_MASK,
 `kRCM_SourceLoc` = RCM_SRS0_LOC_MASK,
 `kRCM_SourceLol` = RCM_SRS0_LOL_MASK,
 `kRCM_SourceWdog` = RCM_SRS0_WDOG_MASK,
 `kRCM_SourcePin` = RCM_SRS0_PIN_MASK,
 `kRCM_SourcePor` = RCM_SRS0 POR_MASK,
 `kRCM_SourceJtag` = RCM_SRS1_JTAG_MASK << 8U,
 `kRCM_SourceLockup` = RCM_SRS1_LOCKUP_MASK << 8U,
 `kRCM_SourceSw` = RCM_SRS1_SW_MASK << 8U,
 `kRCM_SourceMdmap` = RCM_SRS1_MDM_AP_MASK << 8U,
 `kRCM_SourceEzpt` = RCM_SRS1_EZPT_MASK << 8U,
 `kRCM_SourceSackerr` = RCM_SRS1_SACKERR_MASK << 8U }
 System Reset Source Name definitions.
- enum `rcm_run_wait_filter_mode_t` {
 `kRCM_FilterDisable` = 0U,
 `kRCM_FilterBusClock` = 1U,
 `kRCM_FilterLpoClock` = 2U }
 Reset pin filter select in Run and Wait modes.

Driver version

- #define `FSL_RCM_DRIVER_VERSION` (MAKE_VERSION(2, 0, 1))
RCM driver version 2.0.1.

Reset Control Module APIs

- static uint32_t `RCM_GetPreviousResetSources` (RCM_Type *base)
Gets the reset source status which caused a previous reset.

Enumeration Type Documentation

- static uint32_t [RCM_GetStickyResetSources](#) (RCM_Type *base)
Gets the sticky reset source status.
- static void [RCM_ClearStickyResetSources](#) (RCM_Type *base, uint32_t sourceMasks)
Clears the sticky reset source status.
- void [RCM_ConfigureResetPinFilter](#) (RCM_Type *base, const [rcm_reset_pin_filter_config_t](#) *config)
Configures the reset pin filter.
- static bool [RCM_GetEasyPortModePinStatus](#) (RCM_Type *base)
Gets the EZP_MS_B pin assert status.

30.2 Data Structure Documentation

30.2.1 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.

30.2.1.0.0.79 Field Documentation

30.2.1.0.0.79.1 bool rcm_reset_pin_filter_config_t::enableFilterInStop

30.2.1.0.0.79.2 rcm_run_wait_filter_mode_t rcm_reset_pin_filter_config_t::filterInRunWait

30.2.1.0.0.79.3 uint8_t rcm_reset_pin_filter_config_t::busClockFilterCount

30.3 Macro Definition Documentation

30.3.1 #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

30.4 Enumeration Type Documentation

30.4.1 enum rcm_reset_source_t

Enumerator

- kRCM_SourceWakeup* Low-leakage wakeup reset.
- kRCM_SourceLvd* Low-voltage detect reset.
- kRCM_SourceLoc* Loss of clock reset.
- kRCM_SourceLol* Loss of lock reset.
- kRCM_SourceWdog* Watchdog reset.
- kRCM_SourcePin* External pin reset.
- kRCM_SourcePor* Power on reset.
- kRCM_SourceJtag* JTAG generated reset.

kRCM_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceMdmap MDM-AP system reset.
kRCM_SourceEzpt EzPort reset.
kRCM_SourceSackerr Parameter could get all reset flags.

30.4.2 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.

30.5 Function Documentation

30.5.1 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.

Function Documentation

30.5.2 static uint32_t RCM_GetStickyResetSources (RCM_Type * *base*) [inline], [static]

This function gets the current reset source status that has not been cleared by software for a specific source.
This is an example.

```
uint32_t resetStatus;  
  
// To get all reset source statuses.  
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceAll;  
  
// To test whether the MCU is reset using Watchdog.  
resetStatus = RCM_GetStickyResetSources(RCM) &  
    kRCM_SourceWdog;  
  
// To test multiple reset sources.  
resetStatus = RCM_GetStickyResetSources(RCM) &  
    (kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

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

Returns

All reset source status bit map.

30.5.3 static void RCM_ClearStickyResetSources (RCM_Type * *base*, uint32_t *sourceMasks*) [inline], [static]

This function clears the sticky system reset flags indicated by source masks.

This is an example.

```
// Clears multiple reset sources.  
RCM_ClearStickyResetSources(kRCM_SourceWdog |  
    kRCM_SourcePin);
```

Parameters

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

<i>sourceMasks</i>	reset source status bit map
--------------------	-----------------------------

30.5.4 void RCM_ConfigureResetPinFilter (RCM_Type * *base*, const rcm_reset_pin_filter_config_t * *config*)

This function sets the reset pin filter including the filter source, filter width, and so on.

Parameters

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

30.5.5 static bool RCM_GetEasyPortModePinStatus (RCM_Type * *base*) [inline], [static]

This function gets the easy port mode status (EZP_MS_B) pin assert status.

Parameters

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

Returns

status true - asserted, false - reasserted

Function Documentation

Chapter 31

RNGA: Random Number Generator Accelerator Driver

31.1 Overview

The Kinetis SDK provides Peripheral driver for the Random Number Generator Accelerator (RNGA) block of Kinetis devices.

31.2 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.

31.3 Get random data from RNGA

1. [RNGA_GetRandomData\(\)](#) function gets random data from the RNGA module.

31.4 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.

31.5 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:

```
{\n    status_t          status;\n    uint32_t         data;\n\n    /* Initialize RNGA */\n    status = RNGA_Init(RNG);\n\n    /* Read Random data*/\n    status = RNGA_GetRandomData(RNG, data, sizeof(data));\n\n    if(status == kStatus_Success)\n    {\n        /* Print data*/\n        PRINTF("Random = 0x%X\r\n", i, data );\n        PRINTF("Succeed.\r\n");\n    }\n    else\n    {\n\n    }\n}
```

Seed RNGA

```
    PRINTF("RNGA failed! (0x%lx)\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.

31.6 Macro Definition Documentation

31.6.1 #define FSL_RNGA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

31.7 Enumeration Type Documentation

31.7.1 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.

31.8 Function Documentation

31.8.1 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
-------------	-------------------

31.8.2 void RNGA_Deinit (RNG_Type * *base*)

This function shuts down the RNGA.

Parameters

Function Documentation

<i>base</i>	RNGA base address
-------------	-------------------

31.8.3 **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

31.8.4 **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

31.8.5 **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
-------------	---------------------------

31.8.6 **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

Function Documentation

Chapter 32

RTC: Real Time Clock

32.1 Overview

The KSDK provides a driver for the Real Time Clock (RTC) of Kinetis devices.

32.2 Function groups

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

32.2.1 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.

32.2.2 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.

32.2.3 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

32.2.4 Start & Stop timer

The function [RTC_StartTimer\(\)](#) starts the RTC time counter.

The function [RTC_StopTimer\(\)](#) stops the RTC time counter.

32.2.5 Status

Provides functions to get and clear the RTC status.

32.2.6 Interrupt

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

32.2.7 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.

32.2.8 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.

32.3 Typical use case

32.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm.

```
int main(void)
{
    uint32_t sec;
    uint32_t currSeconds;
    rtc_datetime_t date;
    rtc_config_t rtcConfig;

    /* Board pin, clock, debug console init */

    /* Set the RTC current time and trigger an alarm */

    /* ... */

    /* Poll for interrupt */

    /* ... */
}
```

```

BOARD_InitHardware();
/* Init RTC */
RTC_GetDefaultConfig(&rtcConfig);
RTC_Init(RTC, &rtcConfig);
/* Select RTC clock source */
BOARD_SetRtcClockSource();

PRINTF("RTC example: set up time to wake up an alarm\r\n");

/* Set a start date time and start RT */
date.year = 2014U;
date.month = 12U;
date.day = 25U;
date.hour = 19U;
date.minute = 0;
date.second = 0;

/* RTC time counter has to be stopped before setting the date & time in the TSR register */
RTC_StopTimer(RTC);

/* Set RTC time to default */
RTC_SetDatetime(RTC, &date);

/* Enable RTC alarm interrupt */
RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);

/* Enable at the NVIC */
EnableIRQ(RTC IRQn);

/* Start the RTC time counter */
RTC_StartTimer(RTC);

/* This loop will set the RTC alarm */
while (1)
{
    busyWait = true;
    /* Get date time */
    RTC_GetDatetime(RTC, &date);

    /* print default time */
    PRINTF("Current datetime: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day, date.hour,
        date.minute, date.second);

    /* Get alarm time from the user */
    sec = 0;
    PRINTF("Input the number of second to wait for alarm \r\n");
    PRINTF("The second must be positive value\r\n");
    while (sec < 1)
    {
        SCANF("%d", &sec);
    }

    /* Read the RTC seconds register to get current time in seconds */
    currSeconds = RTC->TSR;

    /* Add alarm seconds to current time */
    currSeconds += sec;

    /* Set alarm time in seconds */
    RTC->TAR = currSeconds;

    /* Get alarm time */
    RTC_GetAlarm(RTC, &date);

    /* Print alarm time */
    PRINTF("Alarm will occur at: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day,

```

Typical use case

```
    date.hour, date.minute, date.second);

    /* Wait until alarm occurs */
    while (busyWait)
    {
    }

    PRINTF("\r\n Alarm occurs !!!! ");
}

}
```

Data Structures

- struct `rtc_datetime_t`
Structure is used to hold the date and time. [More...](#)
- struct `rtc_config_t`
RTC config structure. [More...](#)

Enumerations

- enum `rtc_interrupt_enable_t` {
 kRTC_TimeInvalidInterruptEnable = RTC_IER_TIE_MASK,
 kRTC_TimeOverflowInterruptEnable = RTC_IER_TOIE_MASK,
 kRTC_AlarmInterruptEnable = RTC_IER_TAIE_MASK,
 kRTC_SecondsInterruptEnable = RTC_IER_TSIE_MASK }
List of RTC interrupts.
- enum `rtc_status_flags_t` {
 kRTC_TimeInvalidFlag = RTC_SR_TIF_MASK,
 kRTC_TimeOverflowFlag = RTC_SR_TOF_MASK,
 kRTC_AlarmFlag = RTC_SR_TAF_MASK }
List of RTC flags.
- enum `rtc_osc_cap_load_t` {
 kRTC_Capacitor_2p = RTC_CR_SC2P_MASK,
 kRTC_Capacitor_4p = RTC_CR_SC4P_MASK,
 kRTC_Capacitor_8p = RTC_CR_SC8P_MASK,
 kRTC_Capacitor_16p = RTC_CR_SC16P_MASK }
List of RTC Oscillator capacitor load settings.

Functions

- static void `RTC_SetOscCapLoad` (RTC_Type *base, uint32_t capLoad)
This function sets the specified capacitor configuration for the RTC oscillator.
- static void `RTC_Reset` (RTC_Type *base)
Performs a software reset on the RTC module.

Driver version

- #define `FSL_RTC_DRIVER_VERSION` (MAKE_VERSION(2, 0, 0))
Version 2.0.0.

Initialization and deinitialization

- void **RTC_Init** (RTC_Type *base, const **rtc_config_t** *config)
Ungates the RTC clock and configures the peripheral for basic operation.
- static void **RTC_Deinit** (RTC_Type *base)
Stops the timer and gate the RTC clock.
- void **RTC_GetDefaultConfig** (**rtc_config_t** *config)
Fills in the RTC config struct with the default settings.

Current Time & Alarm

- status_t **RTC_SetDatetime** (RTC_Type *base, const **rtc_datetime_t** *datetime)
Sets the RTC date and time according to the given time structure.
- void **RTC_GetDatetime** (RTC_Type *base, **rtc_datetime_t** *datetime)
Gets the RTC time and stores it in the given time structure.
- status_t **RTC_SetAlarm** (RTC_Type *base, const **rtc_datetime_t** *alarmTime)
Sets the RTC alarm time.
- void **RTC_GetAlarm** (RTC_Type *base, **rtc_datetime_t** *datetime)
Returns the RTC alarm time.

Interrupt Interface

- static void **RTC_EnableInterrupts** (RTC_Type *base, uint32_t mask)
Enables the selected RTC interrupts.
- static void **RTC_DisableInterrupts** (RTC_Type *base, uint32_t mask)
Disables the selected RTC interrupts.
- static uint32_t **RTC_GetEnabledInterrupts** (RTC_Type *base)
Gets the enabled RTC interrupts.

Status Interface

- static uint32_t **RTC_GetStatusFlags** (RTC_Type *base)
Gets the RTC status flags.
- void **RTC_ClearStatusFlags** (RTC_Type *base, uint32_t mask)
Clears the RTC status flags.

Timer Start and Stop

- static void **RTC_StartTimer** (RTC_Type *base)
Starts the RTC time counter.
- static void **RTC_StopTimer** (RTC_Type *base)
Stops the RTC time counter.

Monotonic counter functions

- void **RTC_GetMonotonicCounter** (RTC_Type *base, uint64_t *counter)
Reads the values of the Monotonic Counter High and Monotonic Counter Low and returns them as a single value.
- void **RTC_SetMonotonicCounter** (RTC_Type *base, uint64_t counter)
Writes values Monotonic Counter High and Monotonic Counter Low by decomposing the given single value.

Data Structure Documentation

- status_t [RTC_IncrementMonotonicCounter](#) (RTC_Type *base)
Increments the Monotonic Counter by one.

32.4 Data Structure Documentation

32.4.1 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.
- uint8_t [second](#)
Range from 0 to 59.

32.4.1.0.0.80 Field Documentation

32.4.1.0.0.80.1 uint16_t rtc_datetime_t::year

32.4.1.0.0.80.2 uint8_t rtc_datetime_t::month

32.4.1.0.0.80.3 uint8_t rtc_datetime_t::day

32.4.1.0.0.80.4 uint8_t rtc_datetime_t::hour

32.4.1.0.0.80.5 uint8_t rtc_datetime_t::minute

32.4.1.0.0.80.6 uint8_t rtc_datetime_t::second

32.4.2 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.

32.5 Enumeration Type Documentation

32.5.1 enum rtc_interrupt_enable_t

Enumerator

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

32.5.2 enum rtc_status_flags_t

Enumerator

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

32.5.3 enum rtc_osc_cap_load_t

Enumerator

- kRTC_Capacitor_2p*** 2 pF capacitor load
- kRTC_Capacitor_4p*** 4 pF capacitor load
- kRTC_Capacitor_8p*** 8 pF capacitor load
- kRTC_Capacitor_16p*** 16 pF capacitor load

32.6 Function Documentation

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

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

Function Documentation

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.

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

Parameters

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

32.6.3 void RTC_GetDefaultConfig (rtc_config_t * *config*)

The default values are as follows.

```
*     config->wakeupSelect = false;
*     config->updateMode = false;
*     config->supervisorAccess = false;
*     config->compensationInterval = 0;
*     config->compensationTime = 0;
*
```

Parameters

<i>config</i>	Pointer to the user's RTC configuration structure.
---------------	--

32.6.4 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

32.6.5 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.

32.6.6 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

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

Function Documentation

Parameters

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

**32.6.8 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

**32.6.9 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

**32.6.10 static uint32_t RTC_GetEnabledInterrupts (RTC_Type * *base*)
[inline], [static]**

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [rtc_interrupt_enable_t](#)

**32.6.11 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](#)

32.6.12 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

32.6.13 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
-------------	-----------------------------

32.6.14 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
-------------	-----------------------------

32.6.15 static void RTC_SetOscCapLoad (RTC_Type * *base*, uint32_t *capLoad*) [inline], [static]

Function Documentation

Parameters

<i>base</i>	RTC peripheral base address
<i>capLoad</i>	Oscillator loads to enable. This is a logical OR of members of the enumeration rtc_osc_cap_load_t

32.6.16 static void RTC_Reset(RTC_Type * *base*) [inline], [static]

This resets all RTC registers except for the SWR bit and the RTC_WAR and RTC_RAR registers. The SWR bit is cleared by software explicitly clearing it.

Parameters

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

32.6.17 void RTC_GetMonotonicCounter(RTC_Type * *base*, uint64_t * *counter*)

Parameters

<i>base</i>	RTC peripheral base address
<i>counter</i>	Pointer to variable where the value is stored.

32.6.18 void RTC_SetMonotonicCounter(RTC_Type * *base*, uint64_t *counter*)

Parameters

<i>base</i>	RTC peripheral base address
<i>counter</i>	Counter value

32.6.19 status_t RTC_IncrementMonotonicCounter(RTC_Type * *base*)

Increments the Monotonic Counter (registers RTC_MCLR and RTC_MCHR accordingly) by setting the monotonic counter enable (MER[MCE]) and then writing to the RTC_MCLR register. A write to the monotonic counter low that causes it to overflow also increments the monotonic counter high.

Parameters

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

Returns

kStatus_Success: success
kStatus_Fail: error occurred, either time invalid or monotonic overflow flag was found

Function Documentation

Chapter 33

SAI: Serial Audio Interface

33.1 Overview

The KSDK provides a peripheral driver for the Serial Audio Interface (SAI) module of Kinetis devices.

SAI driver includes functional APIs and transactional APIs.

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

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the sai_handle_t as the first parameter. Initialize the handle by calling the [SAI_TransferTxCreateHandle\(\)](#) or [SAI_TransferRxCreateHandle\(\)](#) API.

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

33.2 Typical use case

33.2.1 SAI Send/receive using an interrupt method

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

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

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

void main(void)
{
    //...
    SAI_TxGetDefaultConfig(&user_config);
```

Typical use case

```
SAI_TxInit(SAI0, &user_config);
SAI_TransferTxCreateHandle(SAI0, &g_saiHandle, SAI_UserCallback, NULL);

//Configure sai format
SAI_TransferTxSetTransferFormat(SAI0, &g_saiHandle, mclkSource, mclk);

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

// Send out.
SAI_TransferSendNonBlocking(SAI0, &g_saiHandle, &sendXfer);

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

// ...
}
```

33.2.2 SAI Send/receive using a DMA method

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

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

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

void main(void)
{
    //...

    SAI_TxGetDefaultConfig(&user_config);
    SAI_TxInit(SAI0, &user_config);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, SAI_TX_DMA_CHANNEL, SAI_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SAI_TX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle. */
    DMA_CreateHandle(&g_saiTxDmaHandle, DMA0, SAI_TX_DMA_CHANNEL);

    SAI_TransferTxCreateHandleDMA(SAI0, &g_saiTxDmaHandle, SAI_UserCallback,
        NULL);

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

```

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

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

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

// ...
}

```

Modules

- SAI DMA Driver
- SAI eDMA Driver

Data Structures

- struct `sai_config_t`
`SAI user configuration structure.` [More...](#)
- struct `sai_transfer_format_t`
`sai transfer format` [More...](#)
- struct `sai_transfer_t`
`SAI transfer structure.` [More...](#)
- struct `sai_handle_t`
`SAI handle structure.` [More...](#)

Macros

- #define `SAI_XFER_QUEUE_SIZE` (4)
`SAI transfer queue size, user can refine it according to use case.`

Typedefs

- typedef void(* `sai_transfer_callback_t`)(I2S_Type *base, sai_handle_t *handle, status_t status, void *userData)
`SAI transfer callback prototype.`

Enumerations

- enum `_sai_status_t` {
 `kStatus_SAI_TxBusy` = MAKE_STATUS(kStatusGroup_SAI, 0),
 `kStatus_SAI_RxBusy` = MAKE_STATUS(kStatusGroup_SAI, 1),
 `kStatus_SAI_TxError` = MAKE_STATUS(kStatusGroup_SAI, 2),
 `kStatus_SAI_RxError` = MAKE_STATUS(kStatusGroup_SAI, 3),
 `kStatus_SAI_QueueFull` = MAKE_STATUS(kStatusGroup_SAI, 4),
 `kStatus_SAI_TxIdle` = MAKE_STATUS(kStatusGroup_SAI, 5),
 `kStatus_SAI_RxIdle` = MAKE_STATUS(kStatusGroup_SAI, 6) }
- `SAI return status.`

Typical use case

- enum `sai_protocol_t` {
 `kSAI_BusLeftJustified` = 0x0U,
 `kSAI_BusRightJustified`,
 `kSAI_BusI2S`,
 `kSAI_BusPCMA`,
 `kSAI_BusPCMB` }
 Define the SAI bus type.
- enum `sai_master_slave_t` {
 `kSAI_Master` = 0x0U,
 `kSAI_Slave` = 0x1U }
 Master or slave mode.
- enum `sai_mono_stereo_t` {
 `kSAI_Stereo` = 0x0U,
 `kSAI_MonoLeft`,
 `kSAI_MonoRight` }
 Mono or stereo audio format.
- enum `sai_sync_mode_t` {
 `kSAI_ModeAsync` = 0x0U,
 `kSAI_ModeSync`,
 `kSAI_ModeSyncWithOtherTx`,
 `kSAI_ModeSyncWithOtherRx` }
 Synchronous or asynchronous mode.
- enum `sai_mclk_source_t` {
 `kSAI_MclkSourceSysclk` = 0x0U,
 `kSAI_MclkSourceSelect1`,
 `kSAI_MclkSourceSelect2`,
 `kSAI_MclkSourceSelect3` }
 Mater clock source.
- enum `sai_bclk_source_t` {
 `kSAI_BclkSourceBusclk` = 0x0U,
 `kSAI_BclkSourceMclkDiv`,
 `kSAI_BclkSourceOtherSai0`,
 `kSAI_BclkSourceOtherSai1` }
 Bit clock source.
- enum `_sai_interrupt_enable_t` {
 `kSAI_WordStartInterruptEnable`,
 `kSAI_SyncErrorInterruptEnable` = I2S_TCSR_SEIE_MASK,
 `kSAI_FIFOWarningInterruptEnable` = I2S_TCSR_FWIE_MASK,
 `kSAI_FIFOErrorInterruptEnable` = I2S_TCSR_FEIE_MASK,
 `kSAI_FIFORequestInterruptEnable` = I2S_TCSR_FRIE_MASK }
 The SAI interrupt enable flag.
- enum `_sai_dma_enable_t` {
 `kSAI_FIFOWarningDMAEnable` = I2S_TCSR_FWDE_MASK,
 `kSAI_FIFORequestDMAEnable` = I2S_TCSR_FRDE_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_FIFORequestFlag = I2S_TCSR_FRF_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_fifo_packing_t` {

kSAI_FifoPackingDisabled = 0x0U,

kSAI_FifoPacking8bit = 0x2U,

kSAI_FifoPacking16bit = 0x3U }

The SAI packing mode The mode includes 8 bit and 16 bit packing.

- enum `sai_sample_rate_t` {

kSAI_SampleRate8KHz = 8000U,

kSAI_SampleRate11025Hz = 11025U,

kSAI_SampleRate12KHz = 12000U,

kSAI_SampleRate16KHz = 16000U,

kSAI_SampleRate22050Hz = 22050U,

kSAI_SampleRate24KHz = 24000U,

kSAI_SampleRate32KHz = 32000U,

kSAI_SampleRate44100Hz = 44100U,

kSAI_SampleRate48KHz = 48000U,

kSAI_SampleRate96KHz = 96000U }

Audio sample rate.

- enum `sai_word_width_t` {

kSAI_WordWidth8bits = 8U,

kSAI_WordWidth16bits = 16U,

kSAI_WordWidth24bits = 24U,

kSAI_WordWidth32bits = 32U }

Audio word width.

Driver version

- #define `FSL_SAI_DRIVER_VERSION` (MAKE_VERSION(2, 1, 1))

Version 2.1.1.

Initialization and deinitialization

- void `SAL_TxInit` (I2S_Type *base, const `sai_config_t` *config)

Initializes the SAI Tx peripheral.
- void `SAL_RxInit` (I2S_Type *base, const `sai_config_t` *config)

Initializes the the SAI Rx peripheral.
- void `SAI_TxGetDefaultConfig` (`sai_config_t` *config)

Typical use case

- 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)
 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.

Data Structure Documentation

33.3 Data Structure Documentation

33.3.1 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.
- `bool mclkOutputEnable`
Master clock output enable, true means master clock divider enabled.
- `sai_mclk_source_t mclkSource`
Master Clock source.
- `sai_bclk_source_t bclkSource`
Bit Clock source.
- `sai_master_slave_t masterSlave`
Master or slave.

33.3.2 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 watermark`
Watermark value.
- `uint8_t channel`
Data channel used in transfer.
- `sai_protocol_t protocol`
Which audio protocol used.

33.3.2.0.0.81 Field Documentation

33.3.2.0.0.81.1 uint8_t sai_transfer_format_t::channel

33.3.3 struct sai_transfer_t

Data Fields

- `uint8_t * data`
Data start address to transfer.

- size_t **dataSize**
Transfer size.

33.3.3.0.0.82 Field Documentation

33.3.3.0.0.82.1 uint8_t* **sai_transfer_t::data**

33.3.3.0.0.82.2 size_t **sai_transfer_t::dataSize**

33.3.4 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.
- uint8_t **watermark**
Watermark value.

33.4 Macro Definition Documentation

33.4.1 #define SAI_XFER_QUEUE_SIZE (4)

33.5 Enumeration Type Documentation

33.5.1 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.

Enumeration Type Documentation

kStatus_SAI_TxIdle SAI Tx is idle.

kStatus_SAI_RxIdle SAI Rx is idle.

33.5.2 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.

33.5.3 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode.

kSAI_Slave Slave mode.

33.5.4 enum sai_mono_stereo_t

Enumerator

kSAI_Stereo Stereo sound.

kSAI_MonoLeft Only left channel have sound.

kSAI_MonoRight Only Right channel have sound.

33.5.5 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.

33.5.6 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.

33.5.7 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.

33.5.8 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.
- kSAI_FIFORequestInterruptEnable*** FIFO request, means reached watermark.

33.5.9 enum _sai_dma_enable_t

Enumerator

- kSAI_FIFOWarningDMAEnable*** FIFO warning caused by the DMA request.
- kSAI_FIFORequestDMAEnable*** FIFO request caused by the DMA request.

33.5.10 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.

Enumeration Type Documentation

kSAI_FIFOErrorFlag FIFO error flag.
kSAI_FIFORequestFlag FIFO request flag.
kSAI_FIFOWarningFlag FIFO warning flag.

33.5.11 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.

33.5.12 enum sai_fifo_packing_t

Enumerator

kSAI_FifoPackingDisabled Packing disabled.
kSAI_FifoPacking8bit 8 bit packing enabled
kSAI_FifoPacking16bit 16bit packing enabled

33.5.13 enum sai_sample_rate_t

Enumerator

kSAI_SampleRate8KHz Sample rate 8000 Hz.
kSAI_SampleRate11025Hz Sample rate 11025 Hz.
kSAI_SampleRate12KHz Sample rate 12000 Hz.
kSAI_SampleRate16KHz Sample rate 16000 Hz.
kSAI_SampleRate22050Hz Sample rate 22050 Hz.
kSAI_SampleRate24KHz Sample rate 24000 Hz.
kSAI_SampleRate32KHz Sample rate 32000 Hz.
kSAI_SampleRate44100Hz Sample rate 44100 Hz.
kSAI_SampleRate48KHz Sample rate 48000 Hz.
kSAI_SampleRate96KHz Sample rate 96000 Hz.

33.5.14 enum sai_word_width_t

Enumerator

kSAI_WordWidth8bits Audio data width 8 bits.

- kSAI_WordWidth16bits*** Audio data width 16 bits.
- kSAI_WordWidth24bits*** Audio data width 24 bits.
- kSAI_WordWidth32bits*** Audio data width 32 bits.

33.6 Function Documentation

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

Ungates the SAI clock, resets the module, and configures SAI Tx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI_TxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAIM module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

33.6.2 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.

33.6.3 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);
```

Function Documentation

Parameters

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

33.6.4 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);
```

Parameters

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

33.6.5 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
-------------	------------------

33.6.6 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
-------------	------------------

33.6.7 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
-------------	------------------

33.6.8 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.

33.6.9 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.

33.6.10 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.

33.6.11 static void SAI_TxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Function Documentation

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

33.6.12 static uint32_t SAI_RxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Rx status flag value. Use the Status Mask to get the status value needed.

33.6.13 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

33.6.14 static void SAI_TxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

**33.6.15 static void SAI_RxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

Function Documentation

**33.6.16 static void SAI_TxDisableInterrupts (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

**33.6.17 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

33.6.18 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_FIFOResponseDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

33.6.19 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_FIFOResponseDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

33.6.20 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.

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

33.6.21 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.

Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

33.6.22 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.

33.6.23 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.

33.6.24 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.

33.6.25 static void SAI_WriteData (I2S_Type * *base*, uint32_t *channel*, uint32_t *data*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>data</i>	Data needs to be written.

33.6.26 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.

Function Documentation

33.6.27 **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.

33.6.28 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

33.6.29 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.

Function Documentation

33.6.30 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>mclkSourceClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSourceClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is the status_t.

33.6.31 **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>mclkSourceClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSourceClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is one of status_t.

33.6.32 **status_t SAI_TransferSendNonBlocking (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_t * *xfer*)**

Function Documentation

Note

This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>xfer</i>	Pointer to the sai_transfer_t structure.

Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_TxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

33.6.33 **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 sai_handle_t structure which stores the transfer state.
<i>xfer</i>	Pointer to the sai_transfer_t structure.

Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_RxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

33.6.34 **status_t SAI_TransferGetSendCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)**

Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count sent.

Return values

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

33.6.35 **status_t SAI_TransferGetReceiveCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count received.

Return values

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

33.6.36 **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.

33.6.37 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.

33.6.38 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.

33.6.39 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.

33.7 SAI DMA Driver

33.7.1 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.

33.7.2 Data Structure Documentation

33.7.2.1 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.

33.7.2.1.0.83 Field Documentation

33.7.2.1.0.83.1 `sai_transfer_t sai_dma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

33.7.2.1.0.83.2 `volatile uint8_t sai_dma_handle_t::queueUser`

33.7.3 Function Documentation

33.7.3.1 `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.

33.7.3.2 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.

33.7.3.3 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.

33.7.3.4 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.

33.7.3.5 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.

33.7.3.6 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.

33.7.3.7 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.

33.7.3.8 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.

33.7.3.9 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.

33.7.3.10 status_t SAI_TransferGetReceiveCountDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

SAI DMA Driver

<i>handle</i>	SAI DMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

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

33.8 SAI eDMA Driver

33.8.1 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.

33.8.2 Data Structure Documentation

33.8.2.1 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.

33.8.2.1.0.84 Field Documentation

33.8.2.1.0.84.1 `uint8_t sai_edma_handle_t::nbytes`

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

33.8.2.1.0.84.3 `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

33.8.2.1.0.84.4 `volatile uint8_t sai_edma_handle_t::queueUser`

33.8.3 Function Documentation

33.8.3.1 `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.

33.8.3.2 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.

33.8.3.3 void SAI_TransferTxSetFormatEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSourceClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSourceClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master clock, this value should equals to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

33.8.3.4 void SAI_TransferRxSetFormatEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSourceClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSourceClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is the master clock, this value should equal to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

33.8.3.5 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.

33.8.3.6 **status_t SAI_TransferReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*)**

Note

This interface returns immediately after the transfer initiates. Call the SAI_GetReceiveRemainingBytes to poll the transfer status and check whether the SAI transfer is finished.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA receive successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_RxBusy</i>	SAI is busy receiving data.

33.8.3.7 **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.

33.8.3.8 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.

33.8.3.9 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.

33.8.3.10 status_t SAI_TransferGetReceiveCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

SAI eDMA Driver

<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is no non-blocking transaction in progress.

Chapter 34

SDHC: Secured Digital Host Controller Driver

34.1 Overview

The KSDK provides a peripheral driver for the Secured Digital Host Controller (SDHC) module of Kinetis devices.

34.2 Typical use case

34.2.1 SDHC Operation

```
/* Initializes the SDHC. */
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Fills state in the card driver. */
card->sdhcBase = BOARD_SDHC_BASEADDR;
card->sdhcSourceClock = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->sdhcTransfer = sdhc_transfer_function;

/* Initializes the card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

PRINTF("\r\nRead/Write/Erase the card continuously until it encounters error.....\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);
```

Data Structures

- struct [sdhc_adma2_descriptor_t](#)
Defines the ADMA2 descriptor structure. [More...](#)
- struct [sdhc_capability_t](#)

Typical use case

- *SDHC capability information.* [More...](#)
- struct `sdhc_transfer_config_t`
Card transfer configuration. [More...](#)
- struct `sdhc_boot_config_t`
Data structure to configure the MMC boot feature. [More...](#)
- struct `sdhc_config_t`
Data structure to initialize the SDHC. [More...](#)
- struct `sdhc_data_t`
Card data descriptor. [More...](#)
- struct `sdhc_command_t`
Card command descriptor. [More...](#)
- struct `sdhc_transfer_t`
Transfer state. [More...](#)
- struct `sdhc_transfer_callback_t`
SDHC callback functions. [More...](#)
- struct `sdhc_handle_t`
SDHC handle. [More...](#)
- struct `sdhc_host_t`
SDHC host descriptor. [More...](#)

Macros

- #define `SDHC_MAX_BLOCK_COUNT` (`SDHC_BLKATTR_BLKCNT_MASK >> SDHC_BLKATTR_BLKCNT_SHIFT`)
Maximum block count can be set one time.
- #define `SDHC_ADMA1_ADDRESS_ALIGN` (4096U)
The alignment size for ADDRESS filed in ADMA1's descriptor.
- #define `SDHC_ADMA1_LENGTH_ALIGN` (4096U)
The alignment size for LENGTH field in ADMA1's descriptor.
- #define `SDHC_ADMA2_ADDRESS_ALIGN` (4U)
The alignment size for ADDRESS field in ADMA2's descriptor.
- #define `SDHC_ADMA2_LENGTH_ALIGN` (4U)
The alignment size for LENGTH filed in ADMA2's descriptor.
- #define `SDHC_ADMA1_DESCRIPTOR_ADDRESS_SHIFT` (12U)
The bit shift for ADDRESS filed in ADMA1's descriptor.
- #define `SDHC_ADMA1_DESCRIPTOR_ADDRESS_MASK` (0xFFFFFU)
The bit mask for ADDRESS field in ADMA1's descriptor.
- #define `SDHC_ADMA1_DESCRIPTOR_LENGTH_SHIFT` (12U)
The bit shift for LENGTH filed in ADMA1's descriptor.
- #define `SDHC_ADMA1_DESCRIPTOR_LENGTH_MASK` (0xFFFFU)
The mask for LENGTH field in ADMA1's descriptor.
- #define `SDHC_ADMA1_DESCRIPTOR_MAX_LENGTH_PER_ENTRY` (`SDHC_ADMA1_DESCRIPTOR_LENGTH_MASK + 1U`)
The maximum value of LENGTH filed in ADMA1's descriptor.
- #define `SDHC_ADMA2_DESCRIPTOR_LENGTH_SHIFT` (16U)
The bit shift for LENGTH field in ADMA2's descriptor.
- #define `SDHC_ADMA2_DESCRIPTOR_LENGTH_MASK` (0xFFFFU)
The bit mask for LENGTH field in ADMA2's descriptor.
- #define `SDHC_ADMA2_DESCRIPTOR_MAX_LENGTH_PER_ENTRY` (`SDHC_ADMA2_DESCRIPTOR_LENGTH_MASK`)
The maximum value of LENGTH field in ADMA2's descriptor.

TypeDefs

- `typedef uint32_t sdhc_adma1_descriptor_t`
Defines the adma1 descriptor structure.
- `typedef status_t(* sdhc_transfer_function_t)(SDHC_Type *base, sdhc_transfer_t *content)`
SDHC transfer function.

Enumerations

- `enum _sdhc_status {`
`kStatus_SDHC_BusyTransferring = MAKE_STATUS(kStatusGroup_SDHC, 0U),`
`kStatus_SDHC_PrepareAdmaDescriptorFailed = MAKE_STATUS(kStatusGroup_SDHC, 1U),`
`kStatus_SDHC_SendCommandFailed = MAKE_STATUS(kStatusGroup_SDHC, 2U),`
`kStatus_SDHC_TransferDataFailed = MAKE_STATUS(kStatusGroup_SDHC, 3U) }`
SDHC status.
- `enum _sdhc_capability_flag {`
`kSDHC_SupportAdmaFlag = SDHC_HTCAPBLT_ADMAS_MASK,`
`kSDHC_SupportHighSpeedFlag = SDHC_HTCAPBLT_HSS_MASK,`
`kSDHC_SupportDmaFlag = SDHC_HTCAPBLT_DMAS_MASK,`
`kSDHC_SupportSuspendResumeFlag = SDHC_HTCAPBLT_SRS_MASK,`
`kSDHC_SupportV330Flag = SDHC_HTCAPBLT_VS33_MASK,`
`kSDHC_Support4BitFlag = (SDHC_HTCAPBLT_MBL_SHIFT << 0U),`
`kSDHC_Support8BitFlag = (SDHC_HTCAPBLT_MBL_SHIFT << 1U) }`
Host controller capabilities flag mask.
- `enum _sdhc_wakeup_event {`
`kSDHC_WakeupEventOnCardInt = SDHC_PROCTL_WECINT_MASK,`
`kSDHC_WakeupEventOnCardInsert = SDHC_PROCTL_WECINS_MASK,`
`kSDHC_WakeupEventOnCardRemove = SDHC_PROCTL_WECRM_MASK,`
`kSDHC_WakeupEventsAll }`
Wakeup event mask.
- `enum _sdhc_reset {`
`kSDHC_ResetAll = SDHC_SYSCTL_RSTA_MASK,`
`kSDHC_ResetCommand = SDHC_SYSCTL_RSTC_MASK,`
`kSDHC_ResetData = SDHC_SYSCTL_RSTD_MASK,`
`kSDHC_ResetsAll = (kSDHC_ResetAll | kSDHC_ResetCommand | kSDHC_ResetData) }`
Reset type mask.
- `enum _sdhc_transfer_flag {`

Typical use case

```
kSDHC_EnableDmaFlag = SDHC_XFERTYP_DMAEN_MASK,  
kSDHC_CommandTypeSuspendFlag = (SDHC_XFERTYP_CMDTYP(1U)),  
kSDHC_CommandTypeResumeFlag = (SDHC_XFERTYP_CMDTYP(2U)),  
kSDHC_CommandTypeAbortFlag = (SDHC_XFERTYP_CMDTYP(3U)),  
kSDHC_EnableBlockCountFlag = SDHC_XFERTYP_BCEN_MASK,  
kSDHC_EnableAutoCommand12Flag = SDHC_XFERTYP_AC12EN_MASK,  
kSDHC_DataReadFlag = SDHC_XFERTYP_DTDSEL_MASK,  
kSDHC_MultipleBlockFlag = SDHC_XFERTYP_MSBSEL_MASK,  
kSDHC_ResponseLength136Flag = SDHC_XFERTYP_RSPTYP(1U),  
kSDHC_ResponseLength48Flag = SDHC_XFERTYP_RSPTYP(2U),  
kSDHC_ResponseLength48BusyFlag = SDHC_XFERTYP_RSPTYP(3U),  
kSDHC_EnableCrcCheckFlag = SDHC_XFERTYP_CCCEN_MASK,  
kSDHC_EnableIndexCheckFlag = SDHC_XFERTYP_CICEN_MASK,  
kSDHC_DataPresentFlag = SDHC_XFERTYP_DPSEL_MASK }
```

Transfer flag mask.

- enum _sdhc_present_status_flag {
 kSDHC_CommandInhibitFlag = SDHC_PRSSTAT_CIHB_MASK,
 kSDHC_DataInhibitFlag = SDHC_PRSSTAT_CDIHB_MASK,
 kSDHC_DataLineActiveFlag = SDHC_PRSSTAT_DLA_MASK,
 kSDHC_SdClockStableFlag = SDHC_PRSSTAT_SDSTB_MASK,
 kSDHC_WriteTransferActiveFlag = SDHC_PRSSTAT_WTA_MASK,
 kSDHC_ReadTransferActiveFlag = SDHC_PRSSTAT_RTA_MASK,
 kSDHC_BufferWriteEnableFlag = SDHC_PRSSTAT_BWEN_MASK,
 kSDHC_BufferReadEnableFlag = SDHC_PRSSTAT_BREN_MASK,
 kSDHC_CardInsertedFlag = SDHC_PRSSTAT_CINS_MASK,
 kSDHC_CommandLineLevelFlag = SDHC_PRSSTAT_CLSL_MASK,
 kSDHC_Data0LineLevelFlag = (1U << 24U),
 kSDHC_Data1LineLevelFlag = (1U << 25U),
 kSDHC_Data2LineLevelFlag = (1U << 26U),
 kSDHC_Data3LineLevelFlag = (1U << 27U),
 kSDHC_Data4LineLevelFlag = (1U << 28U),
 kSDHC_Data5LineLevelFlag = (1U << 29U),
 kSDHC_Data6LineLevelFlag = (1U << 30U),
 kSDHC_Data7LineLevelFlag = (1U << 31U) }

Present status flag mask.

- enum _sdhc_interrupt_status_flag {

```

kSDHC_CommandCompleteFlag = SDHC_IRQSTAT_CC_MASK,
kSDHC_DataCompleteFlag = SDHC_IRQSTAT_TC_MASK,
kSDHC_BlockGapEventFlag = SDHC_IRQSTAT_BGE_MASK,
kSDHC_DmaCompleteFlag = SDHC_IRQSTAT_DINT_MASK,
kSDHC_BufferWriteReadyFlag = SDHC_IRQSTAT_BWR_MASK,
kSDHC_BufferReadReadyFlag = SDHC_IRQSTAT_BRR_MASK,
kSDHC_CardInsertionFlag = SDHC_IRQSTAT_CINS_MASK,
kSDHC_CardRemovalFlag = SDHC_IRQSTAT_CRM_MASK,
kSDHC_CardInterruptFlag = SDHC_IRQSTAT_CINT_MASK,
kSDHC_CommandTimeoutFlag = SDHC_IRQSTAT_CTOE_MASK,
kSDHC_CommandCrcErrorFlag = SDHC_IRQSTAT_CCE_MASK,
kSDHC_CommandEndBitErrorFlag = SDHC_IRQSTAT_CEBE_MASK,
kSDHC_CommandIndexErrorFlag = SDHC_IRQSTAT_CIE_MASK,
kSDHC_DataTimeoutFlag = SDHC_IRQSTAT_DTOE_MASK,
kSDHC_DataCrcErrorFlag = SDHC_IRQSTAT_DCE_MASK,
kSDHC_DataEndBitErrorFlag = SDHC_IRQSTAT_DEBE_MASK,
kSDHC_AutoCommand12ErrorFlag = SDHC_IRQSTAT_AC12E_MASK,
kSDHC_DmaErrorFlag = SDHC_IRQSTAT_DMAE_MASK,
kSDHC_CommandErrorFlag,
kSDHC_DataErrorFlag,
kSDHC_ErrorFlag = (kSDHC_CommandErrorFlag | kSDHC_DataErrorFlag | kSDHC_DmaErrorFlag),
kSDHC_DataFlag,
kSDHC_CommandFlag = (kSDHC_CommandErrorFlag | kSDHC_CommandCompleteFlag),
kSDHC_CardDetectFlag = (kSDHC_CardInsertionFlag | kSDHC_CardRemovalFlag),
kSDHC_AllInterruptFlags }

```

Interrupt status flag mask.

- enum _sdhc_auto_command12_error_status_flag {
 kSDHC_AutoCommand12NotExecutedFlag = SDHC_AC12ERR_AC12NE_MASK,
 kSDHC_AutoCommand12TimeoutFlag = SDHC_AC12ERR_AC12TOE_MASK,
 kSDHC_AutoCommand12EndBitErrorFlag = SDHC_AC12ERR_AC12EBE_MASK,
 kSDHC_AutoCommand12CrcErrorFlag = SDHC_AC12ERR_AC12CE_MASK,
 kSDHC_AutoCommand12IndexErrorFlag = SDHC_AC12ERR_AC12IE_MASK,
 kSDHC_AutoCommand12NotIssuedFlag = SDHC_AC12ERR_CNIBAC12E_MASK }

Auto CMD12 error status flag mask.

- enum _sdhc_adma_error_status_flag {
 kSDHC_AdmaLengthMismatchFlag = SDHC_ADMAESADMALME_MASK,
 kSDHC_AdmaDescriptorErrorFlag = SDHC_ADMAESADMADCE_MASK }

ADMA error status flag mask.

- enum sdhc_adma_error_state_t {
 kSDHC_AdmaErrorStateStopDma = 0x00U,
 kSDHC_AdmaErrorStateFetchDescriptor = 0x01U,
 kSDHC_AdmaErrorStateChangeAddress = 0x02U,
 kSDHC_AdmaErrorStateTransferData = 0x03U }

ADMA error state.

- enum _sdhc_force_event {

Typical use case

```
kSDHC_ForceEventAutoCommand12NotExecuted = SDHC_FEVT_AC12NE_MASK,  
kSDHC_ForceEventAutoCommand12Timeout = SDHC_FEVT_AC12TOE_MASK,  
kSDHC_ForceEventAutoCommand12CrcError = SDHC_FEVT_AC12CE_MASK,  
kSDHC_ForceEventEndBitError = SDHC_FEVT_AC12EBE_MASK,  
kSDHC_ForceEventAutoCommand12IndexError = SDHC_FEVT_AC12IE_MASK,  
kSDHC_ForceEventAutoCommand12NotIssued = SDHC_FEVT_CNIBAC12E_MASK,  
kSDHC_ForceEventCommandTimeout = SDHC_FEVT_CTOE_MASK,  
kSDHC_ForceEventCommandCrcError = SDHC_FEVT_CCE_MASK,  
kSDHC_ForceEventCommandEndBitError = SDHC_FEVT_CEBE_MASK,  
kSDHC_ForceEventCommandIndexError = SDHC_FEVT_CIE_MASK,  
kSDHC_ForceEventDataTimeout = SDHC_FEVT_DTOE_MASK,  
kSDHC_ForceEventDataCrcError = SDHC_FEVT_DCE_MASK,  
kSDHC_ForceEventDataEndBitError = SDHC_FEVT_DEBE_MASK,  
kSDHC_ForceEventAutoCommand12Error = SDHC_FEVT_AC12E_MASK,  
kSDHC_ForceEventCardInt = SDHC_FEVT_CINT_MASK,  
kSDHC_ForceEventDmaError = SDHC_FEVT_DMAE_MASK,  
kSDHC_ForceEventsAll }
```

Force event mask.

- enum `sdhc_data_bus_width_t` {
 kSDHC_DataBusWidth1Bit = 0U,
 kSDHC_DataBusWidth4Bit = 1U,
 kSDHC_DataBusWidth8Bit = 2U }

Data transfer width.

- enum `sdhc_endian_mode_t` {
 kSDHC_EndianModeBig = 0U,
 kSDHC_EndianModeHalfWordBig = 1U,
 kSDHC_EndianModeLittle = 2U }

Endian mode.

- enum `sdhc_dma_mode_t` {
 kSDHC_DmaModeNo = 0U,
 kSDHC_DmaModeAdma1 = 1U,
 kSDHC_DmaModeAdma2 = 2U }

DMA mode.

- enum `_sdhc_sdio_control_flag` {
 kSDHC_StopAtBlockGapFlag = 0x01,
 kSDHC_ReadWaitControlFlag = 0x02,
 kSDHC_InterruptAtBlockGapFlag = 0x04,
 kSDHC_ExactBlockNumberReadFlag = 0x08 }

SDIO control flag mask.

- enum `sdhc_boot_mode_t` {
 kSDHC_BootModeNormal = 0U,
 kSDHC_BootModeAlternative = 1U }

MMC card boot mode.

- enum `sdhc_command_type_t` {

```
kSDHC_CommandTypeNormal = 0U,
kSDHC_CommandTypeSuspend = 1U,
kSDHC_CommandTypeResume = 2U,
kSDHC_CommandTypeAbort = 3U }
```

The command type.

- enum `sdhc_response_type_t` {
 kSDHC_ResponseTypeNone = 0U,
 kSDHC_ResponseTypeR1 = 1U,
 kSDHC_ResponseTypeR1b = 2U,
 kSDHC_ResponseTypeR2 = 3U,
 kSDHC_ResponseTypeR3 = 4U,
 kSDHC_ResponseTypeR4 = 5U,
 kSDHC_ResponseTypeR5 = 6U,
 kSDHC_ResponseTypeR5b = 7U,
 kSDHC_ResponseTypeR6 = 8U,
 kSDHC_ResponseTypeR7 = 9U }

The command response type.

- enum `_sdhc_adma1_descriptor_flag` {
 kSDHC_Adma1DescriptorValidFlag = (1U << 0U),
 kSDHC_Adma1DescriptorEndFlag = (1U << 1U),
 kSDHC_Adma1DescriptorInterrupFlag = (1U << 2U),
 kSDHC_Adma1DescriptorActivity1Flag = (1U << 4U),
 kSDHC_Adma1DescriptorActivity2Flag = (1U << 5U),
 kSDHC_Adma1DescriptorTypeNop = (kSDHC_Adma1DescriptorValidFlag),
 kSDHC_Adma1DescriptorTypeTransfer,
 kSDHC_Adma1DescriptorTypeLink,
 kSDHC_Adma1DescriptorTypeSetLength }

The mask for the control/status field in ADMA1 descriptor.

- enum `_sdhc_adma2_descriptor_flag` {
 kSDHC_Adma2DescriptorValidFlag = (1U << 0U),
 kSDHC_Adma2DescriptorEndFlag = (1U << 1U),
 kSDHC_Adma2DescriptorInterruptFlag = (1U << 2U),
 kSDHC_Adma2DescriptorActivity1Flag = (1U << 4U),
 kSDHC_Adma2DescriptorActivity2Flag = (1U << 5U),
 kSDHC_Adma2DescriptorTypeNop = (kSDHC_Adma2DescriptorValidFlag),
 kSDHC_Adma2DescriptorTypeReserved,
 kSDHC_Adma2DescriptorTypeTransfer,
 kSDHC_Adma2DescriptorTypeLink }

ADMA1 descriptor control and status mask.

Driver version

- #define `FSL_SDHC_DRIVER_VERSION` (MAKE_VERSION(2U, 1U, 2U))
Driver version 2.1.2.

Typical use case

Initialization and deinitialization

- void [SDHC_Init](#) (SDHC_Type *base, const [sdhc_config_t](#) *config)
SDHC module initialization function.
- void [SDHC_Deinit](#) (SDHC_Type *base)
Deinitializes the SDHC.
- bool [SDHC_Reset](#) (SDHC_Type *base, uint32_t mask, uint32_t timeout)
Resets the SDHC.

DMA Control

- status_t [SDHC_SetAdmaTableConfig](#) (SDHC_Type *base, [sdhc_dma_mode_t](#) dmaMode, uint32_t *table, uint32_t tableWords, const uint32_t *data, uint32_t dataBytes)
Sets the ADMA descriptor table configuration.

Interrupts

- static void [SDHC_EnableInterruptStatus](#) (SDHC_Type *base, uint32_t mask)
Enables the interrupt status.
- static void [SDHC_DisableInterruptStatus](#) (SDHC_Type *base, uint32_t mask)
Disables the interrupt status.
- static void [SDHC_EnableInterruptSignal](#) (SDHC_Type *base, uint32_t mask)
Enables the interrupt signal corresponding to the interrupt status flag.
- static void [SDHC_DisableInterruptSignal](#) (SDHC_Type *base, uint32_t mask)
Disables the interrupt signal corresponding to the interrupt status flag.

Status

- static uint32_t [SDHC_GetInterruptStatusFlags](#) (SDHC_Type *base)
Gets the current interrupt status.
- static void [SDHC_ClearInterruptStatusFlags](#) (SDHC_Type *base, uint32_t mask)
Clears a specified interrupt status.
- static uint32_t [SDHC_GetAutoCommand12ErrorStatusFlags](#) (SDHC_Type *base)
Gets the status of auto command 12 error.
- static uint32_t [SDHC_GetAdmaErrorStatusFlags](#) (SDHC_Type *base)
Gets the status of the ADMA error.
- static uint32_t [SDHC_GetPresentStatusFlags](#) (SDHC_Type *base)
Gets a present status.

Bus Operations

- void [SDHC_GetCapability](#) (SDHC_Type *base, [sdhc_capability_t](#) *capability)
Gets the capability information.
- static void [SDHC_EnableSdClock](#) (SDHC_Type *base, bool enable)
Enables or disables the SD bus clock.
- uint32_t [SDHC_SetSdClock](#) (SDHC_Type *base, uint32_t srcClock_Hz, uint32_t busClock_Hz)
Sets the SD bus clock frequency.
- bool [SDHC_SetCardActive](#) (SDHC_Type *base, uint32_t timeout)
Sends 80 clocks to the card to set it to the active state.
- static void [SDHC_SetDataBusWidth](#) (SDHC_Type *base, [sdhc_data_bus_width_t](#) width)
Sets the data transfer width.

- void **SDHC_SetTransferConfig** (SDHC_Type *base, const **sdhc_transfer_config_t** *config)
Sets the card transfer-related configuration.
- static uint32_t **SDHC_GetCommandResponse** (SDHC_Type *base, uint32_t index)
Gets the command response.
- static void **SDHC_WriteData** (SDHC_Type *base, uint32_t data)
Fills the the data port.
- static uint32_t **SDHC_ReadData** (SDHC_Type *base)
Retrieves the data from the data port.
- static void **SDHC_EnableWakeupEvent** (SDHC_Type *base, uint32_t mask, bool enable)
Enables or disables a wakeup event in low-power mode.
- static void **SDHC_EnableCardDetecfTest** (SDHC_Type *base, bool enable)
Enables or disables the card detection level for testing.
- static void **SDHC_SetCardDetectTestLevel** (SDHC_Type *base, bool high)
Sets the card detection test level.
- void **SDHC_EnableSdioControl** (SDHC_Type *base, uint32_t mask, bool enable)
Enables or disables the SDIO card control.
- static void **SDHC_SetContinueRequest** (SDHC_Type *base)
Restarts a transaction which has stopped at the block GAP for the SDIO card.
- void **SDHC_SetMmcBootConfig** (SDHC_Type *base, const **sdhc_boot_config_t** *config)
Configures the MMC boot feature.
- static void **SDHC_SetForceEvent** (SDHC_Type *base, uint32_t mask)
Forces generating events according to the given mask.

Transactional

- status_t **SDHC_TransferBlocking** (SDHC_Type *base, uint32_t *admaTable, uint32_t admaTableWords, **sdhc_transfer_t** *transfer)
Transfers the command/data using a blocking method.
- void **SDHC_TransferCreateHandle** (SDHC_Type *base, sdhc_handle_t *handle, const **sdhc_transfer_callback_t** *callback, void *userData)
Creates the SDHC handle.
- status_t **SDHC_TransferNonBlocking** (SDHC_Type *base, sdhc_handle_t *handle, uint32_t *admaTable, uint32_t admaTableWords, **sdhc_transfer_t** *transfer)
Transfers the command/data using an interrupt and an asynchronous method.
- void **SDHC_TransferHandleIRQ** (SDHC_Type *base, sdhc_handle_t *handle)
IRQ handler for the SDHC.

34.3 Data Structure Documentation

34.3.1 struct **sdhc_adma2_descriptor_t**

Data Fields

- uint32_t **attribute**
The control and status field.
- const uint32_t * **address**
The address field.

Data Structure Documentation

34.3.2 struct sdhc_capability_t

Defines a structure to save the capability information of SDHC.

Data Fields

- `uint32_t specVersion`
Specification version.
- `uint32_t vendorVersion`
Vendor version.
- `uint32_t maxBlockLength`
Maximum block length unitized as byte.
- `uint32_t maxBlockCount`
Maximum block count can be set one time.
- `uint32_t flags`
Capability flags to indicate the support information(_sdhc_capability_flag)

34.3.3 struct sdhc_transfer_config_t

Define structure to configure the transfer-related command index/argument flags and data block size/data block numbers. This structure needs to be filled each time a command is sent to the card.

Data Fields

- `size_t dataBlockSize`
Data block size.
- `uint32_t dataBlockCount`
Data block count.
- `uint32_t commandArgument`
Command argument.
- `uint32_t commandIndex`
Command index.
- `uint32_t flags`
Transfer flags(_sdhc_transfer_flag)

34.3.4 struct sdhc_boot_config_t

Data Fields

- `uint32_t ackTimeoutCount`
Timeout value for the boot ACK.
- `sdhc_boot_mode_t bootMode`
Boot mode selection.
- `uint32_t blockCount`

- Stop at block gap value of automatic mode.
- bool [enableBootAck](#)
Enable or disable boot ACK.
- bool [enableBoot](#)
Enable or disable fast boot.
- bool [enableAutoStopAtBlockGap](#)
Enable or disable auto stop at block gap function in boot period.

34.3.4.0.0.85 Field Documentation

34.3.4.0.0.85.1 uint32_t [sdhc_boot_config_t::ackTimeoutCount](#)

The available range is 0 ~ 15.

34.3.4.0.0.85.2 [sdhc_boot_mode_t sdhc_boot_config_t::bootMode](#)

34.3.4.0.0.85.3 uint32_t [sdhc_boot_config_t::blockCount](#)

Available range is 0 ~ 65535.

34.3.5 struct [sdhc_config_t](#)

Data Fields

- bool [cardDetectDat3](#)
Enable DAT3 as card detection pin.
- [sdhc_endian_mode_t endianMode](#)
Endian mode.
- [sdhc_dma_mode_t dmaMode](#)
DMA mode.
- uint32_t [readWatermarkLevel](#)
Watermark level for DMA read operation.
- uint32_t [writeWatermarkLevel](#)
Watermark level for DMA write operation.

34.3.5.0.0.86 Field Documentation

34.3.5.0.0.86.1 uint32_t [sdhc_config_t::readWatermarkLevel](#)

Available range is 1 ~ 128.

34.3.5.0.0.86.2 uint32_t [sdhc_config_t::writeWatermarkLevel](#)

Available range is 1 ~ 128.

Data Structure Documentation

34.3.6 struct sdhc_data_t

Defines a structure to contain data-related attribute. 'enableIgnoreError' is used for the case that upper card driver want to ignore the error event to read/write all the data not to stop read/write immediately when error event happen for example bus testing procedure for MMC card.

Data Fields

- bool `enableAutoCommand12`
Enable auto CMD12.
- bool `enableIgnoreError`
Enable to ignore error event to read/write all the data.
- `size_t blockSize`
Block size.
- `uint32_t blockCount`
Block count.
- `uint32_t * rxData`
Buffer to save data read.
- `const uint32_t * txData`
Data buffer to write.

34.3.7 struct sdhc_command_t

Define card command-related attribute.

Data Fields

- `uint32_t index`
Command index.
- `uint32_t argument`
Command argument.
- `sdhc_command_type_t type`
Command type.
- `sdhc_response_type_t responseType`
Command response type.
- `uint32_t response [4U]`
Response for this command.

34.3.8 struct sdhc_transfer_t

Data Fields

- `sdhc_data_t * data`
Data to transfer.

- `sdhc_command_t * command`
Command to send.

34.3.9 struct sdhc_transfer_callback_t

Data Fields

- `void(* CardInserted)(void)`
Card inserted occurs when DAT3/CD pin is for card detect.
- `void(* CardRemoved)(void)`
Card removed occurs.
- `void(* SdioInterrupt)(void)`
SDIO card interrupt occurs.
- `void(* SdioBlockGap)(void)`
SDIO card stopped at block gap occurs.
- `void(* TransferComplete)(SDHC_Type *base, sdhc_handle_t *handle, status_t status, void *userData)`
Transfer complete callback.

34.3.10 struct _sdhc_handle

SDHC handle typedef.

Defines the structure to save the SDHC state information and callback function. The detailed interrupt status when sending a command or transferring data can be obtained from the interruptFlags field by using the mask defined in `sdhc_interrupt_flag_t`.

Note

All the fields except interruptFlags and transferredWords must be allocated by the user.

Data Fields

- `sdhc_data_t *volatile data`
Data to transfer.
- `sdhc_command_t *volatile command`
Command to send.
- `volatile uint32_t interruptFlags`
Interrupt flags of last transaction.
- `volatile uint32_t transferredWords`
Words transferred by DATAPORT way.
- `sdhc_transfer_callback_t callback`
Callback function.
- `void * userData`
Parameter for transfer complete callback.

Enumeration Type Documentation

34.3.11 struct sdhc_host_t

Data Fields

- SDHC_Type * `base`
SDHC peripheral base address.
- uint32_t `sourceClock_Hz`
SDHC source clock frequency united in Hz.
- `sdhc_config_t config`
SDHC configuration.
- `sdhc_capability_t capability`
SDHC capability information.
- `sdhc_transfer_function_t transfer`
SDHC transfer function.

34.4 Macro Definition Documentation

34.4.1 #define FSL_SDHC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 2U))

34.5 Typedef Documentation

34.5.1 typedef uint32_t sdhc_adma1_descriptor_t

34.5.2 typedef status_t(* sdhc_transfer_function_t)(SDHC_Type *base, sdhc_transfer_t *content)

34.6 Enumeration Type Documentation

34.6.1 enum _sdhc_status

Enumerator

kStatus_SDHC_BusyTransferring Transfer is on-going.

kStatus_SDHC_PrepardAdmaDescriptorFailed Set DMA descriptor failed.

kStatus_SDHC_SendCommandFailed Send command failed.

kStatus_SDHC_TransferDataFailed Transfer data failed.

34.6.2 enum _sdhc_capability_flag

Enumerator

kSDHC_SupportAdmaFlag Support ADMA.

kSDHC_SupportHighSpeedFlag Support high-speed.

kSDHC_SupportDmaFlag Support DMA.

kSDHC_SupportSuspendResumeFlag Support suspend/resume.

kSDHC_SupportV330Flag Support voltage 3.3V.

kSDHC_Support4BitFlag Support 4 bit mode.

kSDHC_Support8BitFlag Support 8 bit mode.

34.6.3 enum _sdhc_wakeup_event

Enumerator

kSDHC_WakeupEventOnCardInt Wakeup on card interrupt.

kSDHC_WakeupEventOnCardInsert Wakeup on card insertion.

kSDHC_WakeupEventOnCardRemove Wakeup on card removal.

kSDHC_WakeupEventsAll All wakeup events.

34.6.4 enum _sdhc_reset

Enumerator

kSDHC_ResetAll Reset all except card detection.

kSDHC_ResetCommand Reset command line.

kSDHC_ResetData Reset data line.

kSDHC_ResetsAll All reset types.

34.6.5 enum _sdhc_transfer_flag

Enumerator

kSDHC_EnableDmaFlag Enable DMA.

kSDHC_CommandTypeSuspendFlag Suspend command.

kSDHC_CommandTypeResumeFlag Resume command.

kSDHC_CommandTypeAbortFlag Abort command.

kSDHC_EnableBlockCountFlag Enable block count.

kSDHC_EnableAutoCommand12Flag Enable auto CMD12.

kSDHC_DataReadFlag Enable data read.

kSDHC_MultipleBlockFlag Multiple block data read/write.

kSDHC_ResponseLength136Flag 136 bit response length

kSDHC_ResponseLength48Flag 48 bit response length

kSDHC_ResponseLength48BusyFlag 48 bit response length with busy status

kSDHC_EnableCrcCheckFlag Enable CRC check.

kSDHC_EnableIndexCheckFlag Enable index check.

kSDHC_DataPresentFlag Data present flag.

Enumeration Type Documentation

34.6.6 enum _sdhc_present_status_flag

Enumerator

kSDHC_CommandInhibitFlag Command inhibit.
kSDHC_DataInhibitFlag Data inhibit.
kSDHC_DataLineActiveFlag Data line active.
kSDHC_SdClockStableFlag SD bus clock stable.
kSDHC_WriteTransferActiveFlag Write transfer active.
kSDHC_ReadTransferActiveFlag Read transfer active.
kSDHC_BufferWriteEnableFlag Buffer write enable.
kSDHC_BufferReadEnableFlag Buffer read enable.
kSDHC_CardInsertedFlag Card inserted.
kSDHC_CommandLineLevelFlag Command line signal level.
kSDHC_Data0LineLevelFlag Data0 line signal level.
kSDHC_Data1LineLevelFlag Data1 line signal level.
kSDHC_Data2LineLevelFlag Data2 line signal level.
kSDHC_Data3LineLevelFlag Data3 line signal level.
kSDHC_Data4LineLevelFlag Data4 line signal level.
kSDHC_Data5LineLevelFlag Data5 line signal level.
kSDHC_Data6LineLevelFlag Data6 line signal level.
kSDHC_Data7LineLevelFlag Data7 line signal level.

34.6.7 enum _sdhc_interrupt_status_flag

Enumerator

kSDHC_CommandCompleteFlag Command complete.
kSDHC_DataCompleteFlag Data complete.
kSDHC_BlockGapEventFlag Block gap event.
kSDHC_DmaCompleteFlag DMA interrupt.
kSDHC_BufferWriteReadyFlag Buffer write ready.
kSDHC_BufferReadReadyFlag Buffer read ready.
kSDHC_CardInsertionFlag Card inserted.
kSDHC_CardRemovalFlag Card removed.
kSDHC_CardInterruptFlag Card interrupt.
kSDHC_CommandTimeoutFlag Command timeout error.
kSDHC_CommandCrcErrorFlag Command CRC error.
kSDHC_CommandEndBitErrorFlag Command end bit error.
kSDHC_CommandIndexErrorFlag Command index error.
kSDHC_DataTimeoutFlag Data timeout error.
kSDHC_DataCrcErrorFlag Data CRC error.
kSDHC_DataEndBitErrorFlag Data end bit error.
kSDHC_AutoCommand12ErrorFlag Auto CMD12 error.

kSDHC_DmaErrorFlag DMA error.
kSDHC_CommandErrorFlag Command error.
kSDHC_DataErrorFlag Data error.
kSDHC_ErrorFlag All error.
kSDHC_DataFlag Data interrupts.
kSDHC_CommandFlag Command interrupts.
kSDHC_CardDetectFlag Card detection interrupts.
kSDHC_AllInterruptFlags All flags mask.

34.6.8 enum _sdhc_auto_command12_error_status_flag

Enumerator

kSDHC_AutoCommand12NotExecutedFlag Not executed error.
kSDHC_AutoCommand12TimeoutFlag Timeout error.
kSDHC_AutoCommand12EndBitErrorFlag End bit error.
kSDHC_AutoCommand12CrcErrorFlag CRC error.
kSDHC_AutoCommand12IndexErrorFlag Index error.
kSDHC_AutoCommand12NotIssuedFlag Not issued error.

34.6.9 enum _sdhc_adma_error_status_flag

Enumerator

kSDHC_AdmaLengthMismatchFlag Length mismatch error.
kSDHC_AdmaDescriptorErrorFlag Descriptor error.

34.6.10 enum sdhc_adma_error_state_t

This state is the detail state when ADMA error has occurred.

Enumerator

kSDHC_AdmaErrorStateStopDma Stop DMA.
kSDHC_AdmaErrorStateFetchDescriptor Fetch descriptor.
kSDHC_AdmaErrorStateChangeAddress Change address.
kSDHC_AdmaErrorStateTransferData Transfer data.

Enumeration Type Documentation

34.6.11 enum _sdhc_force_event

Enumerator

kSDHC_ForceEventAutoCommand12NotExecuted Auto CMD12 not executed error.
kSDHC_ForceEventAutoCommand12Timeout Auto CMD12 timeout error.
kSDHC_ForceEventAutoCommand12CrcError Auto CMD12 CRC error.
kSDHC_ForceEventEndBitError Auto CMD12 end bit error.
kSDHC_ForceEventAutoCommand12IndexError Auto CMD12 index error.
kSDHC_ForceEventAutoCommand12NotIssued Auto CMD12 not issued error.
kSDHC_ForceEventCommandTimeout Command timeout error.
kSDHC_ForceEventCommandCrcError Command CRC error.
kSDHC_ForceEventCommandEndBitError Command end bit error.
kSDHC_ForceEventCommandIndexError Command index error.
kSDHC_ForceEventDataTimeout Data timeout error.
kSDHC_ForceEventDataCrcError Data CRC error.
kSDHC_ForceEventDataEndBitError Data end bit error.
kSDHC_ForceEventAutoCommand12Error Auto CMD12 error.
kSDHC_ForceEventCardInt Card interrupt.
kSDHC_ForceEventDmaError Dma error.
kSDHC_ForceEventsAll All force event flags mask.

34.6.12 enum sdhc_data_bus_width_t

Enumerator

kSDHC_DataBusWidth1Bit 1-bit mode
kSDHC_DataBusWidth4Bit 4-bit mode
kSDHC_DataBusWidth8Bit 8-bit mode

34.6.13 enum sdhc_endian_mode_t

Enumerator

kSDHC_EndianModeBig Big endian mode.
kSDHC_EndianModeHalfWordBig Half word big endian mode.
kSDHC_EndianModeLittle Little endian mode.

34.6.14 enum sdhc_dma_mode_t

Enumerator

kSDHC_DmaModeNo No DMA.

kSDHC_DmaModeAdma1 ADMA1 is selected.
kSDHC_DmaModeAdma2 ADMA2 is selected.

34.6.15 enum _sdhc_sdio_control_flag

Enumerator

kSDHC_StopAtBlockGapFlag Stop at block gap.
kSDHC_ReadWaitControlFlag Read wait control.
kSDHC_InterruptAtBlockGapFlag Interrupt at block gap.
kSDHC_ExactBlockNumberReadFlag Exact block number read.

34.6.16 enum sdhc_boot_mode_t

Enumerator

kSDHC_BootModeNormal Normal boot.
kSDHC_BootModeAlternative Alternative boot.

34.6.17 enum sdhc_command_type_t

Enumerator

kSDHC_CommandTypeNormal Normal command.
kSDHC_CommandTypeSuspend Suspend command.
kSDHC_CommandTypeResume Resume command.
kSDHC_CommandTypeAbort Abort command.

34.6.18 enum sdhc_response_type_t

Define the command response type from card to host controller.

Enumerator

kSDHC_ResponseNone Response type: none.
kSDHC_ResponseR1 Response type: R1.
kSDHC_ResponseR1b Response type: R1b.
kSDHC_ResponseR2 Response type: R2.
kSDHC_ResponseR3 Response type: R3.
kSDHC_ResponseR4 Response type: R4.

Function Documentation

kSDHC_ResponseTyeR5 Response type: R5.
kSDHC_ResponseTyeR5b Response type: R5b.
kSDHC_ResponseTyeR6 Response type: R6.
kSDHC_ResponseTyeR7 Response type: R7.

34.6.19 enum _sdhc_adma1_descriptor_flag

Enumerator

kSDHC_Adma1DescriptorValidFlag Valid flag.
kSDHC_Adma1DescriptorEndFlag End flag.
kSDHC_Adma1DescriptorInterrupFlag Interrupt flag.
kSDHC_Adma1DescriptorActivity1Flag Activity 1 flag.
kSDHC_Adma1DescriptorActivity2Flag Activity 2 flag.
kSDHC_Adma1DescriptorTypeNop No operation.
kSDHC_Adma1DescriptorTypeTransfer Transfer data.
kSDHC_Adma1DescriptorTypeLink Link descriptor.
kSDHC_Adma1DescriptorTypeSetLength Set data length.

34.6.20 enum _sdhc_adma2_descriptor_flag

Enumerator

kSDHC_Adma2DescriptorValidFlag Valid flag.
kSDHC_Adma2DescriptorEndFlag End flag.
kSDHC_Adma2DescriptorInterruptFlag Interrupt flag.
kSDHC_Adma2DescriptorActivity1Flag Activity 1 mask.
kSDHC_Adma2DescriptorActivity2Flag Activity 2 mask.
kSDHC_Adma2DescriptorTypeNop No operation.
kSDHC_Adma2DescriptorTypeReserved Reserved.
kSDHC_Adma2DescriptorTypeTransfer Transfer type.
kSDHC_Adma2DescriptorTypeLink Link type.

34.7 Function Documentation

34.7.1 void SDHC_Init (SDHC_Type * *base*, const sdhc_config_t * *config*)

Configures the SDHC according to the user configuration.

Example:

```
sdhc_config_t config;
config.cardDetectDat3 = false;
config.endianMode = kSDHC_EndianModeLittle;
```

```
config.dmaMode = kSDHC_DmaModeAdma2;
config.readWatermarkLevel = 128U;
config.writeWatermarkLevel = 128U;
SDHC_Init(SDHC, &config);
```

Parameters

<i>base</i>	SDHC peripheral base address.
<i>config</i>	SDHC configuration information.

Return values

<i>kStatus_Success</i>	Operate successfully.
------------------------	-----------------------

34.7.2 void SDHC_Deinit (SDHC_Type * *base*)

Parameters

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

34.7.3 bool SDHC_Reset (SDHC_Type * *base*, uint32_t *mask*, uint32_t *timeout*)

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The reset type mask(_sdhc_reset).
<i>timeout</i>	Timeout for reset.

Return values

<i>true</i>	Reset successfully.
<i>false</i>	Reset failed.

34.7.4 status_t SDHC_SetAdmaTableConfig (SDHC_Type * *base*, sdhc_dma_mode_t *dmaMode*, uint32_t * *table*, uint32_t *tableWords*, const uint32_t * *data*, uint32_t *dataBytes*)

Function Documentation

Parameters

<i>base</i>	SDHC peripheral base address.
<i>dmaMode</i>	DMA mode.
<i>table</i>	ADMA table address.
<i>tableWords</i>	ADMA table buffer length united as Words.
<i>data</i>	Data buffer address.
<i>dataBytes</i>	Data length united as bytes.

Return values

<i>kStatus_OutOfRange</i>	ADMA descriptor table length isn't enough to describe data.
<i>kStatus_Success</i>	Operate successfully.

34.7.5 static void SDHC_EnableInterruptStatus (**SDHC_Type** * *base*, **uint32_t** *mask*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	Interrupt status flags mask(_sdhc_interrupt_status_flag).

34.7.6 static void SDHC_DisableInterruptStatus (**SDHC_Type** * *base*, **uint32_t** *mask*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

34.7.7 static void SDHC_EnableInterruptSignal (**SDHC_Type** * *base*, **uint32_t** *mask*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

34.7.8 static void SDHC_DisableInterruptSignal (**SDHC_Type** * *base*, **uint32_t** *mask*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

34.7.9 static uint32_t SDHC_GetInterruptStatusFlags (**SDHC_Type** * *base*) [inline], [static]

Parameters

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

Returns

Current interrupt status flags mask(_sdhc_interrupt_status_flag).

34.7.10 static void SDHC_ClearInterruptStatusFlags (**SDHC_Type** * *base*, **uint32_t** *mask*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

34.7.11 static uint32_t SDHC_GetAutoCommand12ErrorStatusFlags (**SDHC_Type** * *base*) [inline], [static]

Function Documentation

Parameters

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

Returns

Auto command 12 error status flags mask(_sdhc_auto_command12_error_status_flag).

34.7.12 static uint32_t SDHC_GetAdmaErrorStatusFlags (SDHC_Type * *base*) [inline], [static]

Parameters

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

Returns

ADMA error status flags mask(_sdhc_adma_error_status_flag).

34.7.13 static uint32_t SDHC_GetPresentStatusFlags (SDHC_Type * *base*) [inline], [static]

This function gets the present SDHC's status except for an interrupt status and an error status.

Parameters

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

Returns

Present SDHC's status flags mask(_sdhc_present_status_flag).

34.7.14 void SDHC_GetCapability (SDHC_Type * *base*, sdhc_capability_t * *capability*)

Parameters

<i>base</i>	SDHC peripheral base address.
<i>capability</i>	Structure to save capability information.

**34.7.15 static void SDHC_EnableSdClock (SDHC_Type * *base*, bool *enable*)
[inline], [static]**

Parameters

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

**34.7.16 uint32_t SDHC_SetSdClock (SDHC_Type * *base*, uint32_t *srcClock_Hz*,
uint32_t *busClock_Hz*)**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>srcClock_Hz</i>	SDHC source clock frequency united in Hz.
<i>busClock_Hz</i>	SD bus clock frequency united in Hz.

Returns

The nearest frequency of busClock_Hz configured to SD bus.

34.7.17 bool SDHC_SetCardActive (SDHC_Type * *base*, uint32_t *timeout*)

This function must be called each time the card is inserted to ensure that the card can receive the command correctly.

Parameters

Function Documentation

<i>base</i>	SDHC peripheral base address.
<i>timeout</i>	Timeout to initialize card.

Return values

<i>true</i>	Set card active successfully.
<i>false</i>	Set card active failed.

34.7.18 static void SDHC_SetDataBusWidth (SDHC_Type * *base*, sdhc_data_bus_width_t *width*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>width</i>	Data transfer width.

34.7.19 void SDHC_SetTransferConfig (SDHC_Type * *base*, const sdhc_transfer_config_t * *config*)

This function fills the card transfer-related command argument/transfer flag/data size. The command and data are sent by SDHC after calling this function.

Example:

```
sdhc_transfer_config_t transferConfig;
transferConfig.dataBlockSize = 512U;
transferConfig.dataBlockCount = 2U;
transferConfig.commandArgument = 0x01AAU;
transferConfig.commandIndex = 8U;
transferConfig.flags |= (kSDHC_EnableDmaFlag |
    kSDHC_EnableAutoCommand12Flag |
    kSDHC_MultipleBlockFlag);
SDHC_SetTransferConfig(SDHC, &transferConfig);
```

Parameters

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

<i>config</i>	Command configuration structure.
---------------	----------------------------------

34.7.20 static uint32_t SDHC_GetCommandResponse (SDHC_Type * *base*, uint32_t *index*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>index</i>	The index of response register, range from 0 to 3.

Returns

Response register transfer.

34.7.21 static void SDHC_WriteData (SDHC_Type * *base*, uint32_t *data*) [inline], [static]

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>data</i>	The data about to be sent.

34.7.22 static uint32_t SDHC_ReadData (SDHC_Type * *base*) [inline], [static]

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

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

Returns

The data has been read.

Function Documentation

34.7.23 **static void SDHC_EnableWakeupEvent(SDHC_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	Wakeup events mask(_sdhc_wakeup_event).
<i>enable</i>	True to enable, false to disable.

34.7.24 static void SDHC_EnableCardDetectTest (SDHC_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

34.7.25 static void SDHC_SetCardDetectTestLevel (SDHC_Type * *base*, bool *high*) [inline], [static]

This function sets the card detection test level to indicate whether the card is inserted into the SDHC when DAT[3]/ CD pin is selected as a card detection pin. This function can also assert the pin logic when DAT[3]/CD pin is selected as the card detection pin.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>high</i>	True to set the card detect level to high.

34.7.26 void SDHC_EnableSdioControl (SDHC_Type * *base*, uint32_t *mask*, bool *enable*)

Parameters

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

Function Documentation

<i>mask</i>	SDIO card control flags mask(_sdhc_sdio_control_flag).
<i>enable</i>	True to enable, false to disable.

34.7.27 static void SDHC_SetContinueRequest (SDHC_Type * *base*) [inline], [static]

Parameters

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

34.7.28 void SDHC_SetMmcBootConfig (SDHC_Type * *base*, const sdhc_boot_config_t * *config*)

Example:

```
sdhc_boot_config_t config;
config.ackTimeoutCount = 4;
config.bootMode = kSDHC_BootModeNormal;
config.blockCount = 5;
config.enableBootAck = true;
config.enableBoot = true;
config.enableAutoStopAtBlockGap = true;
SDHC_SetMmcBootConfig(SDHC, &config);
```

Parameters

<i>base</i>	SDHC peripheral base address.
<i>config</i>	The MMC boot configuration information.

34.7.29 static void SDHC_SetForceEvent (SDHC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

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

<i>mask</i>	The force events mask(_sdhc_force_event).
-------------	---

34.7.30 **status_t SDHC_TransferBlocking (SDHC_Type * *base*, uint32_t * *admaTable*, uint32_t *admaTableWords*, sdhc_transfer_t * *transfer*)**

This function waits until the command response/data is received or the SDHC encounters an error by polling the status flag. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

Note

There is no need to call the API 'SDHC_TransferCreateHandle' when calling this API.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>admaTable</i>	ADMA table address, can't be null if transfer way is ADMA1/ADMA2.
<i>admaTable-Words</i>	ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.
<i>transfer</i>	Transfer content.

Return values

<i>kStatus_InvalidArgument</i>	Argument is invalid.
<i>kStatus_SDHC_PrepAdmaDescriptorFailed</i>	Prepare ADMA descriptor failed.
<i>kStatus_SDHC_SendCommandFailed</i>	Send command failed.
<i>kStatus_SDHC_TransferDataFailed</i>	Transfer data failed.
<i>kStatus_Success</i>	Operate successfully.

34.7.31 **void SDHC_TransferCreateHandle (SDHC_Type * *base*, sdhc_handle_t * *handle*, const sdhc_transfer_callback_t * *callback*, void * *userData*)**

Function Documentation

Parameters

<i>base</i>	SDHC peripheral base address.
<i>handle</i>	SDHC handle pointer.
<i>callback</i>	Structure pointer to contain all callback functions.
<i>userData</i>	Callback function parameter.

34.7.32 `status_t SDHC_TransferNonBlocking (SDHC_Type * base, sdhc_handle_t * handle, uint32_t * admaTable, uint32_t admaTableWords, sdhc_transfer_t * transfer)`

This function sends a command and data and returns immediately. It doesn't wait the transfer complete or encounter an error. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

Note

Call the API 'SDHC_TransferCreateHandle' when calling this API.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>handle</i>	SDHC handle.
<i>admaTable</i>	ADMA table address, can't be null if transfer way is ADMA1/ADMA2.
<i>admaTable-Words</i>	ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.
<i>transfer</i>	Transfer content.

Return values

<i>kStatus_InvalidArgument</i>	Argument is invalid.
<i>kStatus_SDHC_Busy-Transferring</i>	Busy transferring.

<i>kStatus_SDHC_Prep-AdmaDescriptorFailed</i>	Prepare ADMA descriptor failed.
<i>kStatus_Success</i>	Operate successfully.

34.7.33 void SDHC_TransferHandleIRQ (**SDHC_Type** * *base*, **sdhc_handle_t** * *handle*)

This function deals with the IRQs on the given host controller.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>handle</i>	SDHC handle.

Function Documentation

Chapter 35

SDRAMC: Synchronous DRAM Controller Driver

35.1 Overview

The KSDK provides a peripheral driver for the Synchronous DRAM Controller block of Kinetis devices.

The SDRAM controller commands include the initialization MRS command, precharge command, enter/exit self-refresh command, and enable/disable auto-refresh command. Use the [SDRAMC_SendCommand\(\)](#) to send these commands to SDRAM to initialize it. The [SDRAMC_EnableWriteProtect\(\)](#) is provided to enable/disable the write protection. The [SDRAMC_EnableOperateValid\(\)](#) is provided to enable/disable the operation valid.

35.2 Typical use case

This example shows how to use the SDRAM Controller driver to initialize the external 16 bit port-size 8-column SDRAM chip. Initialize the SDRAM controller and run the initialization sequence. The external SDRAM is initialized and the SDRAM read and write is available.

First, initialize the SDRAM Controller.

```
sdramc_config_t config;
uint32_t clockSrc;

// SDRAM refresh timing configuration.
clockSrc = CLOCK_GetFreq(kCLOCK_BusClk);
sdramc_refresh_config_t refConfig =
{
    kSDRAMC_RefreshThreeClocks,
    15625,      // SDRAM: 4096 rows/ 64ms.
    clockSrc,
};

// SDRAM controller configuration.
sdramc_blockctl_config_t ctrlConfig =
{
    kSDRAMC_Block0,
    kSDRAMC_PortSize16Bit,
    kSDRAMC_Commandbit19,
    kSDRAMC_LatencyOne,
    SDRAM_START_ADDRESS,
    0x7c0000,
};

config.refreshConfig = &refConfig;
config.blockConfig = &ctrlConfig;
config.numBlockConfig = 1;

// SDRAM controller initialization.
SDRAMC_Init(base, &config);
```

Then, run the initialization sequence.

```
// Issues a PALL command.
```

Typical use case

```
SDRAMC_SendCommand(base, whichBlock, kSDRAMC_PrechargeCommand);

// Accesses an SDRAM location.
*(uint8_t *) (SDRAM_START_ADDRESS) = SDRAM_COMMAND_ACCESSVALUE;

// Enables the refresh.
SDRAMC_SendCommand(base, whichBlock,
    kSDRAMC_AutoRefreshEnableCommand);

// Waits for 8 refresh cycles less than one microsecond.
delay;

// Issues the MSR command.
SDRAMC_SendCommand(base, whichBlock, kSDRAMC_ImrsCommand);

// Puts the correct value on the SDRAM address bus for the SDRAM mode register.
addr = ....;

// Set MRS register.
mrsAddr = (uint8_t *) (SDRAM_START_ADDRESS + addr);
*mrsAddr = SDRAM_COMMAND_ACCESSVALUE;
```

Data Structures

- struct `sdrmc_blockctl_config_t`
SDRAM controller block control configuration structure. [More...](#)
- struct `sdrmc_refresh_config_t`
SDRAM controller refresh timing configuration structure. [More...](#)
- struct `sdrmc_config_t`
SDRAM controller configuration structure. [More...](#)

Enumerations

- enum `sdrmc_refresh_time_t`{
 kSDRAMC_RefreshThreeClocks = 0x0U,
 kSDRAMC_RefreshSixClocks,
 kSDRAMC_RefreshNineClocks }
SDRAM controller auto-refresh timing.
- enum `sdrmc_latency_t`{
 kSDRAMC_LatencyZero = 0x0U,
 kSDRAMC_LatencyOne,
 kSDRAMC_LatencyTwo,
 kSDRAMC_LatencyThree }
Setting latency for SDRAM controller timing specifications.
- enum `sdrmc_command_bit_location_t`{
 kSDRAMC_Commandbit17 = 0x0U,
 kSDRAMC_Commandbit18,
 kSDRAMC_Commandbit19,
 kSDRAMC_Commandbit20,
 kSDRAMC_Commandbit21,
 kSDRAMC_Commandbit22,
 kSDRAMC_Commandbit23,
 kSDRAMC_Commandbit24 }
SDRAM controller command bit location.

- enum `sdrmc_command_t` {

 `kSDRAMC_ImrsCommand` = 0x0U,

 `kSDRAMC_PreloadCommand`,

 `kSDRAMC_SelfrefreshEnterCommand`,

 `kSDRAMC_SelfrefreshExitCommand`,

 `kSDRAMC_AutoRefreshEnableCommand`,

 `kSDRAMC_AutoRefreshDisableCommand` }

SDRAM controller command.

- enum `sdrmc_port_size_t` {

 `kSDRAMC_PortSize32Bit` = 0x0U,

 `kSDRAMC_PortSize8Bit`,

 `kSDRAMC_PortSize16Bit` }

SDRAM port size.

- enum `sdrmc_block_selection_t` {

 `kSDRAMC_Block0` = 0x0U,

 `kSDRAMC_Block1` }

SDRAM controller block selection.

Driver version

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

SDRAM Controller Initialization and De-initialization

- void `SDRAMC_Init` (`SDRAM_Type` *base, `sdrmc_config_t` *configure)

Initializes the SDRAM controller.

- void `SDRAMC_Deinit` (`SDRAM_Type` *base)

Deinitializes the SDRAM controller module and gates the clock.

SDRAM Controller Basic Operation

- void `SDRAMC_SendCommand` (`SDRAM_Type` *base, `sdrmc_block_selection_t` block, `sdrmc_command_t` command)

Sends the SDRAM command.

- static void `SDRAMC_EnableWriteProtect` (`SDRAM_Type` *base, `sdrmc_block_selection_t` block, bool enable)

Enables/disables the write protection.

- static void `SDRAMC_EnableOperateValid` (`SDRAM_Type` *base, `sdrmc_block_selection_t` block, bool enable)

Enables/disables the valid operation.

Data Structure Documentation

35.3 Data Structure Documentation

35.3.1 struct sramc_blockctl_config_t

Data Fields

- [sramc_block_selection_t block](#)
The block number.
- [sramc_port_size_t portSize](#)
The port size of the associated SDRAM block.
- [sramc_command_bit_location_t location](#)
The command bit location.
- [sramc_latency_t latency](#)
The latency for some timing specifications.
- [uint32_t address](#)
The base address of the SDRAM block.
- [uint32_t addressMask](#)
The base address mask of the SDRAM block.

35.3.1.0.0.87 Field Documentation

35.3.1.0.0.87.1 sramc_block_selection_t sramc_blockctl_config_t::block

35.3.1.0.0.87.2 sramc_port_size_t sramc_blockctl_config_t::portSize

35.3.1.0.0.87.3 sramc_command_bit_location_t sramc_blockctl_config_t::location

35.3.1.0.0.87.4 sramc_latency_t sramc_blockctl_config_t::latency

35.3.1.0.0.87.5 uint32_t sramc_blockctl_config_t::address

35.3.1.0.0.87.6 uint32_t sramc_blockctl_config_t::addressMask

35.3.2 struct sramc_refresh_config_t

Data Fields

- [sramc_refresh_time_t refreshTime](#)
Trc: The number of bus clocks inserted between a REF and next ACTIVE command.
- [uint32_t sramRefreshRow](#)
The SDRAM refresh time each row: ns/row.
- [uint32_t busClock_Hz](#)
The bus clock for SDRAMC.

35.3.2.0.0.88 Field Documentation**35.3.2.0.0.88.1** `sdramc_refresh_time_t sdramc_refresh_config_t::refreshTime`**35.3.2.0.0.88.2** `uint32_t sdramc_refresh_config_t::sdramRefreshRow`**35.3.2.0.0.88.3** `uint32_t sdramc_refresh_config_t::busClock_Hz`**35.3.3 struct sdramc_config_t**

Defines a configure structure and uses the SDRAMC_Configure() function to make necessary initializations.

Data Fields

- `sdramc_refresh_config_t * refreshConfig`
Refresh timing configure structure pointer.
- `sdramc_blockctl_config_t * blockConfig`
Block configure structure pointer.
- `uint8_t numBlockConfig`
SDRAM block numbers for configuration.

35.3.3.0.0.89 Field Documentation**35.3.3.0.0.89.1** `sdramc_refresh_config_t* sdramc_config_t::refreshConfig`**35.3.3.0.0.89.2** `sdramc_blockctl_config_t* sdramc_config_t::blockConfig`

If both SDRAM blocks are used, use the two continuous blockConfig.

35.3.3.0.0.89.3 `uint8_t sdramc_config_t::numBlockConfig`**35.4 Macro Definition Documentation****35.4.1** `#define FSL_SDRAMC_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`**35.5 Enumeration Type Documentation****35.5.1 enum sdramc_refresh_time_t**

Enumerator

kSDRAMC_RefreshThreeClocks The refresh timing with three bus clocks.

kSDRAMC_RefreshSixClocks The refresh timing with six bus clocks.

kSDRAMC_RefreshNineClocks The refresh timing with nine bus clocks.

Enumeration Type Documentation

35.5.2 enum sdramc_latency_t

The latency setting affects the following SDRAM timing specifications:

- trcd: SRAS assertion to SCAS assertion
- tcasl: SCAS assertion to data out
- tras: ACTV command to Precharge command
- trp: Precharge command to ACTV command
- trwl, trdl: Last data input to Precharge command
- tep: Last data out to Precharge command

The details of the latency setting and timing specifications are shown in the following table list.

latency trcd: tcasl tras trp trwl,trdl tep

0 1 bus clock 1 bus clock 2 bus clocks 1 bus clock 1 bus clock 1 bus clock 1 bus clock

1 2 bus clock 2 bus clock 4 bus clocks 2 bus clock 1 bus clock 1 bus clock 1 bus clock

2 3 bus clock 3 bus clock 6 bus clocks 3 bus clock 1 bus clock 1 bus clock 1 bus clock

3 3 bus clock 3 bus clock 6 bus clocks 3 bus clock 1 bus clock 1 bus clock 1 bus clock

Enumerator

kSDRAMC_LatencyZero Latency 0.

kSDRAMC_LatencyOne Latency 1.

kSDRAMC_LatencyTwo Latency 2.

kSDRAMC_LatencyThree Latency 3.

35.5.3 enum sdramc_command_bit_location_t

Enumerator

kSDRAMC_Commandbit17 Command bit location is bit 17.

kSDRAMC_Commandbit18 Command bit location is bit 18.

kSDRAMC_Commandbit19 Command bit location is bit 19.

kSDRAMC_Commandbit20 Command bit location is bit 20.

kSDRAMC_Commandbit21 Command bit location is bit 21.

kSDRAMC_Commandbit22 Command bit location is bit 22.

kSDRAMC_Commandbit23 Command bit location is bit 23.

kSDRAMC_Commandbit24 Command bit location is bit 24.

35.5.4 enum sdramc_command_t

Enumerator

kSDRAMC_ImrsCommand Initiate MRS command.

kSDRAMC_PrefetchCommand Initiate precharge command.

kSDRAMC_SelfrefreshEnterCommand Enter self-refresh command.

kSDRAMC_SelfrefreshExitCommand Exit self-refresh command.

kSDRAMC_AutoRefreshEnableCommand Enable Auto refresh command.

kSDRAMC_AutoRefreshDisableCommand Disable Auto refresh command.

35.5.5 enum sdrmc_port_size_t

Enumerator

kSDRAMC_PortSize32Bit 32-Bit port size.

kSDRAMC_PortSize8Bit 8-Bit port size.

kSDRAMC_PortSize16Bit 16-Bit port size.

35.5.6 enum sdrmc_block_selection_t

Enumerator

kSDRAMC_Block0 Select SDRAM block 0.

kSDRAMC_Block1 Select SDRAM block 1.

35.6 Function Documentation

35.6.1 void SDRAMC_Init (**SDRAM_Type** * *base*, **sdrmc_config_t** * *configure*)

This function ungates the SDRAM controller clock and initializes the SDRAM controller. This function must be called before calling any other SDRAM controller driver functions. Example

```
sdrmc_refresh_config_t refreshConfig;
sdrmc_blockctl_config_t blockConfig;
sdrmc_config_t config;

refreshConfig.refreshTime = kSDRAM_RefreshThreeClocks;
refreshConfig.sdramRefreshRow = 15625;
refreshConfig.busClock = 60000000;

blockConfig.block = kSDRAMC_Block0;
blockConfig.portSize = kSDRAMC_PortSize16Bit;
blockConfig.location = kSDRAMC_Commandbit19;
blockConfig.latency = kSDRAMC_RefreshThreeClocks;
blockConfig.address = SDRAM_START_ADDRESS;
blockConfig.addressMask = 0x7c0000;

config.refreshConfig = &refreshConfig,
config.blockConfig = &blockConfig,
config.totalBlocks = 1;

SDRAMC_Init (SDRAM, &config);
```

Function Documentation

Parameters

<i>base</i>	SDRAM controller peripheral base address.
<i>configure</i>	The SDRAM configuration structure pointer.

35.6.2 void SDRAMC_Deinit (SDRAM_Type * *base*)

This function gates the SDRAM controller clock. As a result, the SDRAM controller module doesn't work after calling this function.

Parameters

<i>base</i>	SDRAM controller peripheral base address.
-------------	---

35.6.3 void SDRAMC_SendCommand (SDRAM_Type * *base*, sdramc_block_selection_t *block*, sdramc_command_t *command*)

This function sends commands to SDRAM. The commands are precharge command, initialization MRS command, auto-refresh enable/disable command, and self-refresh enter/exit commands. Note that the self-refresh enter/exit commands are all blocks setting and "block" is ignored. Ensure to set the correct "block" when send other commands.

Parameters

<i>base</i>	SDRAM controller peripheral base address.
<i>block</i>	The block selection.
<i>command</i>	The SDRAM command, see "sdramc_command_t". kSDRAMC_ImrsCommand - Initialize MRS command kSDRAMC_PrefetchCommand - Initialize precharge command kSDRAMC_SelfrefreshEnterCommand - Enter self-refresh command kSDRAMC_SelfrefreshExitCommand - Exit self-refresh command kSDRAMC_AutoRefreshEnableCommand - Enable auto refresh command kSDRAMC_AutoRefreshDisableCommand - Disable auto refresh command

35.6.4 static void SDRAMC_EnableWriteProtect (SDRAM_Type * *base*, sdramc_block_selection_t *block*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SDRAM peripheral base address.
<i>block</i>	The block which is selected.
<i>enable</i>	True enable write protection, false disable write protection.

**35.6.5 static void SDRAMC_EnableOperateValid (SDRAM_Type * *base*,
sdramc_block_selection_t *block*, bool *enable*) [inline], [static]**

Parameters

<i>base</i>	SDRAM peripheral base address.
<i>block</i>	The block which is selected.
<i>enable</i>	True enable the valid operation; false disable the valid operation.

Function Documentation

Chapter 36

SIM: System Integration Module Driver

36.1 Overview

The KSDK provides a peripheral driver for the System Integration Module (SIM) of Kinetis devices.

Data Structures

- struct `sim_uid_t`
Unique ID. [More...](#)

Enumerations

- enum `_sim_usb_volt_reg_enable_mode` {
 `kSIM_UsbVoltRegEnable` = SIM_SOPT1_USBREGEN_MASK,
 `kSIM_UsbVoltRegEnableInLowPower` = SIM_SOPT1_USBVSTBY_MASK,
 `kSIM_UsbVoltRegEnableInStop` = SIM_SOPT1_USBSSTBY_MASK,
 `kSIM_UsbVoltRegEnableInAllModes` }
USB voltage regulator enable setting.
- enum `_sim_flash_mode` {
 `kSIM_FlashDisableInWait` = SIM_FCFG1_FLASHDOZE_MASK,
 `kSIM_FlashDisable` = SIM_FCFG1_FLASHDIS_MASK }
Flash enable mode.

Functions

- void `SIM_SetUsbVoltRegulatorEnableMode` (uint32_t mask)
Sets the USB voltage regulator setting.
- void `SIM_GetUniqueId` (`sim_uid_t *uid`)
Gets the unique identification register value.
- static void `SIM_SetFlashMode` (uint8_t mode)
Sets the flash enable mode.

Driver version

- #define `FSL_SIM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)
Driver version 2.0.0.

36.2 Data Structure Documentation

36.2.1 struct `sim_uid_t`

Data Fields

- uint32_t `MH`

Function Documentation

- *UIDMH.*
 • `uint32_t ML`
 UIDML.
 • `uint32_t L`
 UIDL.

36.2.1.0.0.90 Field Documentation

36.2.1.0.0.90.1 `uint32_t sim_uid_t::MH`

36.2.1.0.0.90.2 `uint32_t sim_uid_t::ML`

36.2.1.0.0.90.3 `uint32_t sim_uid_t::L`

36.3 Enumeration Type Documentation

36.3.1 `enum_sim_usb_volt_reg_enable_mode`

Enumerator

kSIM_UsbVoltRegEnable Enable voltage regulator.

kSIM_UsbVoltRegEnableInLowPower Enable voltage regulator in VLPR/VLPW modes.

kSIM_UsbVoltRegEnableInStop Enable voltage regulator in STOP/VLPS/LLS/VLLS modes.

kSIM_UsbVoltRegEnableInAllModes Enable voltage regulator in all power modes.

36.3.2 `enum_sim_flash_mode`

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.

kSIM_FlashDisable Disable flash in normal mode.

36.4 Function Documentation

36.4.1 `void SIM_SetUsbVoltRegulatorEnableMode (uint32_t mask)`

This function configures whether the USB voltage regulator is enabled in normal RUN mode, STOP-/VLPS/LLS/VLLS modes, and VLPR/VLPW modes. The configurations are passed in as mask value of `_sim_usb_volt_reg_enable_mode`. For example, to enable USB voltage regulator in RUN/VLPR/VLPW modes and disable in STOP/VLPS/LLS/VLLS mode, use:

```
SIM_SetUsbVoltRegulatorEnableMode(kSIM_UsbVoltRegEnable | kSIM_UsbVoltRegEnableInLowPower);
```

Parameters

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

36.4.2 void SIM_GetUniqueId (sim_uid_t * *uid*)

Parameters

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

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

Parameters

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

Function Documentation

Chapter 37

SMC: System Mode Controller Driver

37.1 Overview

The KSDK provides a peripheral driver for the System Mode Controller (SMC) module of Kinetis devices. The SMC module sequences the system in and out of all low-power stop and run modes.

API functions are provided to configure the system for working in a dedicated power mode. For different power modes, SMC_SetPowerModeXXX() function accepts different parameters. System power mode state transitions are not available between power modes. For details about available transitions, see the power mode transitions section in the SoC reference manual.

37.2 Typical use case

37.2.1 Enter wait or stop modes

SMC driver provides APIs to set MCU to different wait modes and stop modes. Pre and post functions are used for setting the modes. The pre functions and post functions are used as follows.

1. Disable/enable the interrupt through PRIMASK. This is an example use case. The application sets the wakeup interrupt and calls SMC function `SMC_SetPowerModeStop` to set the MCU to STOP mode, but the wakeup interrupt happens so quickly that the ISR completes before the function `SMC_SetPowerModeStop`. As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMASK. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.
2. Disable/enable the flash speculation. When entering stop modes, the flash speculation might be interrupted. As a result, pre functions disable the flash speculation and post functions enable it.

```
SMC_PreEnterStopModes();  
/* Enable the wakeup interrupt here. */  
SMC_SetPowerModeStop(SMC, kSMC_PartialStop);  
SMC_PostExitStopModes();
```

Data Structures

- struct `smc_power_mode_lls_config_t`
SMC Low-Leakage Stop power mode configuration. [More...](#)
- struct `smc_power_mode_vlls_config_t`
SMC Very Low-Leakage Stop power mode configuration. [More...](#)

Typical use case

Enumerations

- enum `smc_power_mode_protection_t` {
 `kSMC_AllowPowerModeVlls` = SMC_PMPROT_AVLLS_MASK,
 `kSMC_AllowPowerModeLls` = SMC_PMPROT_ALLS_MASK,
 `kSMC_AllowPowerModeVlp` = SMC_PMPROT_AVLP_MASK,
 `kSMC_AllowPowerModeHsrun` = SMC_PMPROT_AHSRUN_MASK,
 `kSMC_AllowPowerModeAll` }
 Power Modes Protection.
- enum `smc_power_state_t` {
 `kSMC_PowerStateRun` = 0x01U << 0U,
 `kSMC_PowerStateStop` = 0x01U << 1U,
 `kSMC_PowerStateVlpr` = 0x01U << 2U,
 `kSMC_PowerStateVlpw` = 0x01U << 3U,
 `kSMC_PowerStateVlps` = 0x01U << 4U,
 `kSMC_PowerStateLls` = 0x01U << 5U,
 `kSMC_PowerStateVlls` = 0x01U << 6U,
 `kSMC_PowerStateHsrun` = 0x01U << 7U }
 Power Modes in PMSTAT.
- enum `smc_run_mode_t` {
 `kSMC_RunNormal` = 0U,
 `kSMC_RunVlpr` = 2U,
 `kSMC_Hsrun` = 3U }
 Run mode definition.
- enum `smc_stop_mode_t` {
 `kSMC_StopNormal` = 0U,
 `kSMC_StopVlps` = 2U,
 `kSMC_StopLls` = 3U,
 `kSMC_StopVlls` = 4U }
 Stop mode definition.
- enum `smc_stop_submode_t` {
 `kSMC_StopSub0` = 0U,
 `kSMC_StopSub1` = 1U,
 `kSMC_StopSub2` = 2U,
 `kSMC_StopSub3` = 3U }
 VLLS/LLS stop sub mode definition.
- enum `smc_partial_stop_option_t` {
 `kSMC_PartialStop` = 0U,
 `kSMC_PartialStop1` = 1U,
 `kSMC_PartialStop2` = 2U }
 Partial STOP option.
- enum `_smc_status` { `kStatus_SMC_StopAbort` = MAKE_STATUS(kStatusGroup_POWER, 0) }
 SMC configuration status.

Driver version

- #define `FSL_SMC_DRIVER_VERSION` (MAKE_VERSION(2, 0, 3))
 SMC driver version 2.0.3.

System mode controller APIs

- static void [SMC_SetPowerModeProtection](#) (SMC_Type *base, uint8_t allowedModes)
Configures all power mode protection settings.
- static smc_power_state_t [SMC_GetPowerModeState](#) (SMC_Type *base)
Gets the current power mode status.
- void [SMC_PreEnterStopModes](#) (void)
Prepares to enter stop modes.
- void [SMC_PostExitStopModes](#) (void)
Recoveries after wake up from stop modes.
- static void [SMC_PreEnterWaitModes](#) (void)
Prepares to enter wait modes.
- static void [SMC_PostExitWaitModes](#) (void)
Recoveries after wake up from wait modes.
- status_t [SMC_SetPowerModeRun](#) (SMC_Type *base)
Configures the system to RUN power mode.
- status_t [SMC_SetPowerModeHsrun](#) (SMC_Type *base)
Configures the system to HSRUN power mode.
- status_t [SMC_SetPowerModeWait](#) (SMC_Type *base)
Configures the system to WAIT power mode.
- status_t [SMC_SetPowerModeStop](#) (SMC_Type *base, [smc_partial_stop_option_t](#) option)
Configures the system to Stop power mode.
- status_t [SMC_SetPowerModeVlpr](#) (SMC_Type *base)
Configures the system to VLPR power mode.
- status_t [SMC_SetPowerModeVlpw](#) (SMC_Type *base)
Configures the system to VLPW power mode.
- status_t [SMC_SetPowerModeVlps](#) (SMC_Type *base)
Configures the system to VLPS power mode.
- status_t [SMC_SetPowerModeLls](#) (SMC_Type *base, const [smc_power_mode_lls_config_t](#) *config)
Configures the system to LLS power mode.
- status_t [SMC_SetPowerModeVlls](#) (SMC_Type *base, const [smc_power_mode_vlls_config_t](#) *config)
Configures the system to VLLS power mode.

37.3 Data Structure Documentation

37.3.1 struct smc_power_mode_lls_config_t

Data Fields

- [smc_stop_submode_t](#) subMode
Low-leakage Stop sub-mode.

37.3.2 struct smc_power_mode_vlls_config_t

Data Fields

- [smc_stop_submode_t](#) subMode

Enumeration Type Documentation

- *Very Low-leakage Stop sub-mode.*
• bool `enablePorDetectInVlls0`
Enable Power on reset detect in VLLS mode.
• bool `enableRam2InVlls2`
Enable RAM2 power in VLLS2.

37.4 Macro Definition Documentation

37.4.1 #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

37.5 Enumeration Type Documentation

37.5.1 enum smc_power_mode_protection_t

Enumerator

- `kSMC_AllowPowerModeVlls` Allow Very-low-leakage Stop Mode.
- `kSMC_AllowPowerModeLls` Allow Low-leakage Stop Mode.
- `kSMC_AllowPowerModeVlp` Allow Very-Low-power Mode.
- `kSMC_AllowPowerModeHsrun` Allow High-speed Run mode.
- `kSMC_AllowPowerModeAll` Allow all power mode.

37.5.2 enum smc_power_state_t

Enumerator

- `kSMC_PowerStateRun` 0000_0001 - Current power mode is RUN
- `kSMC_PowerStateStop` 0000_0010 - Current power mode is STOP
- `kSMC_PowerStateVlpr` 0000_0100 - Current power mode is VLPR
- `kSMC_PowerStateVlpw` 0000_1000 - Current power mode is VLPW
- `kSMC_PowerStateVlps` 0001_0000 - Current power mode is VLPS
- `kSMC_PowerStateLls` 0010_0000 - Current power mode is LLS
- `kSMC_PowerStateVlls` 0100_0000 - Current power mode is VLLS
- `kSMC_PowerStateHsrun` 1000_0000 - Current power mode is HSRUN

37.5.3 enum smc_run_mode_t

Enumerator

- `kSMC_RunNormal` Normal RUN mode.
- `kSMC_RunVlpr` Very-low-power RUN mode.
- `kSMC_Hsrun` High-speed Run mode (HSRUN).

37.5.4 enum smc_stop_mode_t

Enumerator

- kSMC_StopNormal* Normal STOP mode.
- kSMC_StopVlps* Very-low-power STOP mode.
- kSMC_StopLls* Low-leakage Stop mode.
- kSMC_StopVlls* Very-low-leakage Stop mode.

37.5.5 enum smc_stop_submode_t

Enumerator

- kSMC_StopSub0* Stop submode 0, for VLLS0/LLS0.
- kSMC_StopSub1* Stop submode 1, for VLLS1/LLS1.
- kSMC_StopSub2* Stop submode 2, for VLLS2/LLS2.
- kSMC_StopSub3* Stop submode 3, for VLLS3/LLS3.

37.5.6 enum smc_partial_stop_option_t

Enumerator

- kSMC_PartialStop* STOP - Normal Stop mode.
- kSMC_PartialStop1* Partial Stop with both system and bus clocks disabled.
- kSMC_PartialStop2* Partial Stop with system clock disabled and bus clock enabled.

37.5.7 enum _smc_status

Enumerator

- kStatus_SMC_StopAbort* Entering Stop mode is abort.

37.6 Function Documentation

37.6.1 static void SMC_SetPowerModeProtection (**SMC_Type** * *base*, **uint8_t allowedModes**) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the **smc_power_mode_protection_t**. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

Function Documentation

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.

37.6.2 static smc_power_state_t SMC_GetPowerModeState (SMC_Type * *base*) [inline], [static]

This function returns the current power mode status. After the application switches the power mode, it should always check the status to check whether it runs into the specified mode or not. The application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the smc_power_state_t for information about the power status.

Parameters

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

Returns

Current power mode status.

37.6.3 void SMC_PreEnterStopModes (void)

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

37.6.4 void SMC_PostExitStopModes (void)

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

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

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

37.6.6 static void SMC_PostExitWaitModes (void) [inline], [static]

This function should be called after wake up from WAIT/VLPW modes. It is used with [SMC_PreEnterWaitModes](#).

Function Documentation

37.6.7 **status_t SMC_SetPowerModeRun (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

37.6.8 status_t SMC_SetPowerModeHsrun (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

37.6.9 status_t SMC_SetPowerModeWait (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

37.6.10 status_t SMC_SetPowerModeStop (SMC_Type * *base*, smc_partial_stop_option_t *option*)

Parameters

Function Documentation

<i>base</i>	SMC peripheral base address.
<i>option</i>	Partial Stop mode option.

Returns

SMC configuration error code.

37.6.11 status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

37.6.12 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

37.6.13 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

37.6.14 status_t SMC_SetPowerModeLls (SMC_Type * *base*, const smc_power_mode_lls_config_t * *config*)

Parameters

<i>base</i>	SMC peripheral base address.
<i>config</i>	The LLS power mode configuration structure

Returns

SMC configuration error code.

37.6.15 **status_t SMC_SetPowerModeVlls (SMC_Type * *base*, const smc_power_mode_vlls_config_t * *config*)**

Parameters

<i>base</i>	SMC peripheral base address.
<i>config</i>	The VLLS power mode configuration structure.

Returns

SMC configuration error code.

Function Documentation

Chapter 38

TPM: Timer PWM Module

38.1 Overview

The KSDK provides a driver for the Timer PWM Module (TPM) of Kinetis devices.

The KSDK TPM driver supports the generation of PWM signals, input capture, and output compare modes. On some SoCs, the driver supports the generation of combined PWM signals, dual-edge capture, and quadrature decoder modes. The driver also supports configuring each of the TPM fault inputs. The fault input is available only on some SoCs.

The function [TPM_Init\(\)](#) initializes the TPM with a specified configurations. The function [TPM_GetDefaultConfig\(\)](#) gets the default configurations. On some SoCs, the initialization function issues a software reset to reset the TPM internal logic. The initialization function configures the TPM's behavior when it receives a trigger input and its operation in doze and debug modes.

The function [TPM_Deinit\(\)](#) disables the TPM counter and turns off the module clock.

The function [TPM_SetupPwm\(\)](#) sets up TPM channels for the PWM output. The function can set up the PWM signal properties for multiple channels. Each channel has its own [tpm_chnl_pwm_signal_param_t](#) structure that is used to specify the output signals duty cycle and level-mode. However, the same PWM period and PWM mode is applied to all channels requesting a PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 where 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle). When generating a combined PWM signal, the channel number passed refers to a channel pair number, for example 0 refers to channel 0 and 1, 1 refers to channels 2 and 3.

The function [TPM_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular TPM channel.

The function [TPM_UpdateChnlEdgeLevelSelect\(\)](#) updates the level select bits of a particular TPM channel. This can be used to disable the PWM output when making changes to the PWM signal.

The function [TPM_SetupInputCapture\(\)](#) sets up a TPM channel for input capture. The user can specify the capture edge.

The function [TPM_SetupDualEdgeCapture\(\)](#) can be used to measure the pulse width of a signal. This is available only for certain SoCs. A channel pair is used during the capture with the input signal coming through a channel that can be configured. The user can specify the capture edge for each channel and any filter value to be used when processing the input signal.

The function [TPM_SetupOutputCompare\(\)](#) sets up a TPM channel for output comparison. The user can specify the channel output on a successful comparison and a comparison value.

The function [TPM_SetupQuadDecode\(\)](#) sets up TPM channels 0 and 1 for quad decode, which is available only for certain SoCs. The user can specify the quad decode mode, polarity, and filter properties for each input signal.

Typical use case

The function TPM_SetupFault() sets up the properties for each fault, which is available only for certain SoCs. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

Provides functions to get and clear the TPM status.

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

38.2 Typical use case

38.2.1 PWM output

Output the PWM signal on 2 TPM channels with different duty cycles. Periodically update the PWM signal duty cycle.

```
int main(void)
{
    bool brightnessUp = true; /* Indicates whether the LED is brighter or dimmer. */
    tpm_config_t tpmInfo;
    uint8_t updatedDutycycle = 0U;
    tpm_chnl_pwm_signal_param_t tpmParam[2];

    /* Configures the TPM parameters with frequency 24 kHz. */
    tpmParam[0].chnlNumber = (tpm_chnl_t)BOARD_FIRST_TPM_CHANNEL;
    tpmParam[0].level = kTPM_LowTrue;
    tpmParam[0].dutyCyclePercent = 0U;

    tpmParam[1].chnlNumber = (tpm_chnl_t)BOARD_SECOND_TPM_CHANNEL;
    tpmParam[1].level = kTPM_LowTrue;
    tpmParam[1].dutyCyclePercent = 0U;

    /* Board pin, clock, and debug console initialization. */
    BOARD_InitHardware();

    TPM_GetDefaultConfig(&tpmInfo);
    /* Initializes the TPM module. */
    TPM_Init(BOARD_TPM_BASEADDR, &tpmInfo);

    TPM_SetupPwm(BOARD_TPM_BASEADDR, tpmParam, 2U,
                 kTPM_EdgeAlignedPwm, 24000U, TPM_SOURCE_CLOCK);
    TPM_StartTimer(BOARD_TPM_BASEADDR, kTPM_SystemClock);
    while (1)
    {
        /* Delays to see the change of LED brightness. */
        delay();

        if (brightnessUp)
        {
            /* Increases a duty cycle until it reaches a limited value. */
            if (++updatedDutycycle == 100U)
            {
                brightnessUp = false;
            }
        }
        else
        {
            /* Decreases a duty cycle until it reaches a limited value. */
            if (--updatedDutycycle == 0U)
            {
                brightnessUp = true;
            }
        }
    }
}
```

```

    /* Starts PWM mode with an updated duty cycle. */
    TPM_UpdatePwmDutycycle(BOARD_TPM_BASEADDR, (
        tpm_chnl_t)BOARD_FIRST_TPM_CHANNEL, kTPM_EdgeAlignedPwm,
        updatedDutycycle);
    TPM_UpdatePwmDutycycle(BOARD_TPM_BASEADDR, (
        tpm_chnl_t)BOARD_SECOND_TPM_CHANNEL, kTPM_EdgeAlignedPwm,
        updatedDutycycle);
}
}

```

Data Structures

- struct `tpm_chnl_pwm_signal_param_t`
Options to configure a TPM channel's PWM signal. [More...](#)
- struct `tpm_dual_edge_capture_param_t`
TPM dual edge capture parameters. [More...](#)
- struct `tpm_phase_params_t`
TPM quadrature decode phase parameters. [More...](#)
- struct `tpm_config_t`
TPM config structure. [More...](#)

Enumerations

- enum `tpm_chnl_t` {
 kTPM_Chnl_0 = 0U,
 kTPM_Chnl_1,
 kTPM_Chnl_2,
 kTPM_Chnl_3,
 kTPM_Chnl_4,
 kTPM_Chnl_5,
 kTPM_Chnl_6,
 kTPM_Chnl_7 }

List of TPM channels.
- enum `tpm_pwm_mode_t` {
 kTPM_EdgeAlignedPwm = 0U,
 kTPM_CenterAlignedPwm,
 kTPM_CombinedPwm }

TPM PWM operation modes.
- enum `tpm_pwm_level_select_t` {
 kTPM_NoPwmSignal = 0U,
 kTPM_LowTrue,
 kTPM_HighTrue }

TPM PWM output pulse mode: high-true, low-true or no output.
- enum `tpm_trigger_select_t`

Trigger options available.
- enum `tpm_trigger_source_t` {
 kTPM_TriggerSource_External = 0U,
 kTPM_TriggerSource_Internal }

Trigger source options available.
- enum `tpm_output_compare_mode_t` {

Typical use case

- ```
kTPM_NoOutputSignal = (1U << TPM_CnSC_MSA_SHIFT),
kTPM_ToggleOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_S-
HIFT)),
kTPM_ClearOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_SH-
IFT)),
kTPM_SetOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (3U << TPM_CnSC_ELSA_SHIF-
T)),
kTPM_HighPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_-
SHIFT)),
kTPM_LowPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_S-
HIFT)) }
```
- TPM output compare modes.*
- enum `tpm_input_capture_edge_t` {  
  kTPM\_RisingEdge = (1U << TPM\_CnSC\_ELSA\_SHIFT),  
  kTPM\_FallingEdge = (2U << TPM\_CnSC\_ELSA\_SHIFT),  
  kTPM\_RiseAndFallEdge = (3U << TPM\_CnSC\_ELSA\_SHIFT) }
- TPM input capture edge.*
- enum `tpm_quad_decode_mode_t` {  
  kTPM\_QuadPhaseEncode = 0U,  
  kTPM\_QuadCountAndDir }
- TPM quadrature decode modes.*
- enum `tpm_phase_polarity_t` {  
  kTPM\_QuadPhaseNormal = 0U,  
  kTPM\_QuadPhaseInvert }
- TPM quadrature phase polarities.*
- enum `tpm_clock_source_t` {  
  kTPM\_SystemClock = 1U,  
  kTPM\_ExternalClock }
- TPM clock source selection.*
- enum `tpm_clock_prescale_t` {  
  kTPM\_Prescale\_Divide\_1 = 0U,  
  kTPM\_Prescale\_Divide\_2,  
  kTPM\_Prescale\_Divide\_4,  
  kTPM\_Prescale\_Divide\_8,  
  kTPM\_Prescale\_Divide\_16,  
  kTPM\_Prescale\_Divide\_32,  
  kTPM\_Prescale\_Divide\_64,  
  kTPM\_Prescale\_Divide\_128 }
- TPM prescale value selection for the clock source.*
- enum `tpm_interrupt_enable_t` {

```
kTPM_Chnl0InterruptEnable = (1U << 0),
kTPM_Chnl1InterruptEnable = (1U << 1),
kTPM_Chnl2InterruptEnable = (1U << 2),
kTPM_Chnl3InterruptEnable = (1U << 3),
kTPM_Chnl4InterruptEnable = (1U << 4),
kTPM_Chnl5InterruptEnable = (1U << 5),
kTPM_Chnl6InterruptEnable = (1U << 6),
kTPM_Chnl7InterruptEnable = (1U << 7),
kTPM_TimeOverflowInterruptEnable = (1U << 8) }
```

*List of TPM interrupts.*

- enum `tpm_status_flags_t` {
 

```
kTPM_Chnl0Flag = (1U << 0),
kTPM_Chnl1Flag = (1U << 1),
kTPM_Chnl2Flag = (1U << 2),
kTPM_Chnl3Flag = (1U << 3),
kTPM_Chnl4Flag = (1U << 4),
kTPM_Chnl5Flag = (1U << 5),
kTPM_Chnl6Flag = (1U << 6),
kTPM_Chnl7Flag = (1U << 7),
kTPM_TimeOverflowFlag = (1U << 8) }
```

*List of TPM flags.*

## Driver version

- #define `FSL TPM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 2)`)  
*Version 2.0.2.*

## Initialization and deinitialization

- void `TPM_Init` (`TPM_Type *base`, const `tpm_config_t *config`)
 

*Ungates the TPM clock and configures the peripheral for basic operation.*
- void `TPM_Deinit` (`TPM_Type *base`)
 

*Stops the counter and gates the TPM clock.*
- void `TPM_GetDefaultConfig` (`tpm_config_t *config`)
 

*Fill in the TPM config struct with the default settings.*

## Channel mode operations

- status\_t `TPM_SetupPwm` (`TPM_Type *base`, const `tpm_chnl_pwm_signal_param_t *chnlParams`, `uint8_t numOfChnls`, `tpm_pwm_mode_t mode`, `uint32_t pwmFreq_Hz`, `uint32_t srcClock_Hz`)
 

*Configures the PWM signal parameters.*
- void `TPM_UpdatePwmDutyCycle` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `tpm_pwm_mode_t currentPwmMode`, `uint8_t dutyCyclePercent`)
 

*Update the duty cycle of an active PWM signal.*
- void `TPM_UpdateChnlEdgeLevelSelect` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `uint8_t level`)
 

*Update the edge level selection for a channel.*
- void `TPM_SetupInputCapture` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `tpm_input_capture_edge_t captureMode`)

## Data Structure Documentation

- void [TPM\\_SetupOutputCompare](#) (TPM\_Type \*base, [tpm\\_chnl\\_t](#) chnlNumber, [tpm\\_output\\_compare\\_mode\\_t](#) compareMode, uint32\_t compareValue)  
*Enables capturing an input signal on the channel using the function parameters.*  
Configures the TPM to generate timed pulses.
- void [TPM\\_SetupDualEdgeCapture](#) (TPM\_Type \*base, [tpm\\_chnl\\_t](#) chnlPairNumber, const [tpm\\_dual\\_edge\\_capture\\_param\\_t](#) \*edgeParam, uint32\_t filterValue)  
*Configures the dual edge capture mode of the TPM.*
- void [TPM\\_SetupQuadDecode](#) (TPM\_Type \*base, const [tpm\\_phase\\_params\\_t](#) \*phaseAParams, const [tpm\\_phase\\_params\\_t](#) \*phaseBParams, [tpm\\_quad\\_decode\\_mode\\_t](#) quadMode)  
*Configures the parameters and activates the quadrature decode mode.*

## Interrupt Interface

- void [TPM\\_EnableInterrupts](#) (TPM\_Type \*base, uint32\_t mask)  
*Enables the selected TPM interrupts.*
- void [TPM\\_DisableInterrupts](#) (TPM\_Type \*base, uint32\_t mask)  
*Disables the selected TPM interrupts.*
- uint32\_t [TPM\\_GetEnabledInterrupts](#) (TPM\_Type \*base)  
*Gets the enabled TPM interrupts.*

## Status Interface

- static uint32\_t [TPM\\_GetStatusFlags](#) (TPM\_Type \*base)  
*Gets the TPM status flags.*
- static void [TPM\\_ClearStatusFlags](#) (TPM\_Type \*base, uint32\_t mask)  
*Clears the TPM status flags.*

## Timer Start and Stop

- static void [TPM\\_StartTimer](#) (TPM\_Type \*base, [tpm\\_clock\\_source\\_t](#) clockSource)  
*Starts the TPM counter.*
- static void [TPM\\_StopTimer](#) (TPM\_Type \*base)  
*Stops the TPM counter.*

## 38.3 Data Structure Documentation

### 38.3.1 struct tpm\_chnl\_pwm\_signal\_param\_t

#### Data Fields

- [tpm\\_chnl\\_t](#) chnlNumber  
*TPM channel to configure.*
- [tpm\\_pwm\\_level\\_select\\_t](#) level  
*PWM output active level select.*
- uint8\_t dutyCyclePercent  
*PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...*
- uint8\_t firstEdgeDelayPercent  
*Used only in combined PWM mode to generate asymmetrical PWM.*

**38.3.1.0.0.91 Field Documentation****38.3.1.0.0.91.1 tpm\_chnl\_t tpm\_chnl\_pwm\_signal\_param\_t::chnlNumber**

In combined mode (available in some SoC's, this represents the channel pair number

**38.3.1.0.0.91.2 uint8\_t tpm\_chnl\_pwm\_signal\_param\_t::dutyCyclePercent**

100=always active signal (100% duty cycle)

**38.3.1.0.0.91.3 uint8\_t tpm\_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 percentage of the PWM period

**38.3.2 struct tpm\_dual\_edge\_capture\_param\_t**

Note

This mode is available only on some SoC's.

**Data Fields**

- bool [enableSwap](#)  
*true: Use channel n+1 input, channel n input is ignored; false: Use channel n input, channel n+1 input is ignored*
- [tpm\\_input\\_capture\\_edge\\_t currChanEdgeMode](#)  
*Input capture edge select for channel n.*
- [tpm\\_input\\_capture\\_edge\\_t nextChanEdgeMode](#)  
*Input capture edge select for channel n+1.*

**38.3.3 struct tpm\_phase\_params\_t****Data Fields**

- uint32\_t [phaseFilterVal](#)  
*Filter value, filter is disabled when the value is zero.*
- [tpm\\_phase\\_polarity\\_t phasePolarity](#)  
*Phase polarity.*

**38.3.4 struct tpm\_config\_t**

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the [TPM\\_GetDefaultConfig\(\)](#) function and pass a pointer to your config structure

## Enumeration Type Documentation

instance.

The config struct can be made const so it resides in flash

## Data Fields

- **tpm\_clock\_prescale\_t prescale**  
*Select TPM clock prescale value.*
- **bool useGlobalTimeBase**  
*true: Use of an external global time base is enabled; false: disabled*
- **tpm\_trigger\_select\_t triggerSelect**  
*Input trigger to use for controlling the counter operation.*
- **tpm\_trigger\_source\_t triggerSource**  
*Decides if we use external or internal trigger.*
- **bool enableDoze**  
*true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode*
- **bool enableDebugMode**  
*true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode*
- **bool enableReloadOnTrigger**  
*true: TPM counter is reloaded on trigger; false: TPM counter not reloaded*
- **bool enableStopOnOverflow**  
*true: TPM counter stops after overflow; false: TPM counter continues running after overflow*
- **bool enableStartOnTrigger**  
*true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately*
- **bool enablePauseOnTrigger**  
*true: TPM counter will pause while trigger remains asserted; false: TPM counter continues running*

### 38.3.4.0.0.92 Field Documentation

#### 38.3.4.0.0.92.1 tpm\_trigger\_source\_t tpm\_config\_t::triggerSource

## 38.4 Enumeration Type Documentation

### 38.4.1 enum tpm\_chnl\_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kTPM\_Chnl\_0** TPM channel number 0.
- kTPM\_Chnl\_1** TPM channel number 1.
- kTPM\_Chnl\_2** TPM channel number 2.
- kTPM\_Chnl\_3** TPM channel number 3.
- kTPM\_Chnl\_4** TPM channel number 4.
- kTPM\_Chnl\_5** TPM channel number 5.
- kTPM\_Chnl\_6** TPM channel number 6.
- kTPM\_Chnl\_7** TPM channel number 7.

### 38.4.2 enum tpm\_pwm\_mode\_t

Enumerator

*kTPM\_EdgeAlignedPwm* Edge aligned PWM.  
*kTPM\_CenterAlignedPwm* Center aligned PWM.  
*kTPM\_CombinedPwm* Combined PWM.

### 38.4.3 enum tpm\_pwm\_level\_select\_t

Enumerator

*kTPM\_NoPwmSignal* No PWM output on pin.  
*kTPM\_LowTrue* Low true pulses.  
*kTPM\_HighTrue* High true pulses.

### 38.4.4 enum tpm\_trigger\_select\_t

This is used for both internal & external trigger sources (external option available in certain SoC's)

Note

The actual trigger options available is SoC-specific.

### 38.4.5 enum tpm\_trigger\_source\_t

Note

This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal trigger.

Enumerator

*kTPM\_TriggerSource\_External* Use external trigger input.  
*kTPM\_TriggerSource\_Internal* Use internal trigger.

### 38.4.6 enum tpm\_output\_compare\_mode\_t

Enumerator

*kTPM\_NoOutputSignal* No channel output when counter reaches CnV.

## Enumeration Type Documentation

*kTPM\_ToggleOnMatch* Toggle output.  
*kTPM\_ClearOnMatch* Clear output.  
*kTPM\_SetOnMatch* Set output.  
*kTPM\_HighPulseOutput* Pulse output high.  
*kTPM\_LowPulseOutput* Pulse output low.

### 38.4.7 enum tpm\_input\_capture\_edge\_t

Enumerator

*kTPM\_RisingEdge* Capture on rising edge only.  
*kTPM\_FallingEdge* Capture on falling edge only.  
*kTPM\_RiseAndFallEdge* Capture on rising or falling edge.

### 38.4.8 enum tpm\_quad\_decode\_mode\_t

Note

This mode is available only on some SoC's.

Enumerator

*kTPM\_QuadPhaseEncode* Phase A and Phase B encoding mode.  
*kTPM\_QuadCountAndDir* Count and direction encoding mode.

### 38.4.9 enum tpm\_phase\_polarity\_t

Enumerator

*kTPM\_QuadPhaseNormal* Phase input signal is not inverted.  
*kTPM\_QuadPhaseInvert* Phase input signal is inverted.

### 38.4.10 enum tpm\_clock\_source\_t

Enumerator

*kTPM\_SystemClock* System clock.  
*kTPM\_ExternalClock* External clock.

### 38.4.11 enum tpm\_clock\_prescale\_t

Enumerator

- kTPM\_Prescale\_Divide\_1* Divide by 1.
- kTPM\_Prescale\_Divide\_2* Divide by 2.
- kTPM\_Prescale\_Divide\_4* Divide by 4.
- kTPM\_Prescale\_Divide\_8* Divide by 8.
- kTPM\_Prescale\_Divide\_16* Divide by 16.
- kTPM\_Prescale\_Divide\_32* Divide by 32.
- kTPM\_Prescale\_Divide\_64* Divide by 64.
- kTPM\_Prescale\_Divide\_128* Divide by 128.

### 38.4.12 enum tpm\_interrupt\_enable\_t

Enumerator

- kTPM\_Chnl0InterruptEnable* Channel 0 interrupt.
- kTPM\_Chnl1InterruptEnable* Channel 1 interrupt.
- kTPM\_Chnl2InterruptEnable* Channel 2 interrupt.
- kTPM\_Chnl3InterruptEnable* Channel 3 interrupt.
- kTPM\_Chnl4InterruptEnable* Channel 4 interrupt.
- kTPM\_Chnl5InterruptEnable* Channel 5 interrupt.
- kTPM\_Chnl6InterruptEnable* Channel 6 interrupt.
- kTPM\_Chnl7InterruptEnable* Channel 7 interrupt.
- kTPM\_TimeOverflowInterruptEnable* Time overflow interrupt.

### 38.4.13 enum tpm\_status\_flags\_t

Enumerator

- kTPM\_Chnl0Flag* Channel 0 flag.
- kTPM\_Chnl1Flag* Channel 1 flag.
- kTPM\_Chnl2Flag* Channel 2 flag.
- kTPM\_Chnl3Flag* Channel 3 flag.
- kTPM\_Chnl4Flag* Channel 4 flag.
- kTPM\_Chnl5Flag* Channel 5 flag.
- kTPM\_Chnl6Flag* Channel 6 flag.
- kTPM\_Chnl7Flag* Channel 7 flag.
- kTPM\_TimeOverflowFlag* Time overflow flag.

## 38.5 Function Documentation

### 38.5.1 void TPM\_Init ( *TPM\_Type* \* *base*, *const tpm\_config\_t* \* *config* )

## Function Documentation

### Note

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

### Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | TPM peripheral base address             |
| <i>config</i> | Pointer to user's TPM config structure. |

### 38.5.2 void TPM\_Deinit ( **TPM\_Type** \* *base* )

### Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

### 38.5.3 void TPM\_GetDefaultConfig ( **tpm\_config\_t** \* *config* )

The default values are:

```
* config->prescale = kTPM_Prescale_Divide_1;
* config->useGlobalTimeBase = false;
* config->dozeEnable = false;
* config->dbgMode = false;
* config->enableReloadOnTrigger = false;
* config->enableStopOnOverflow = false;
* config->enableStartOnTrigger = false;
*#if FSL_FEATURE TPM HAS_PAUSE_COUNTER_ON_TRIGGER
* config->enablePauseOnTrigger = false;
#endif
* config->triggerSelect = kTPM_Trigger_Select_0;
*#if FSL_FEATURE TPM HAS_EXTERNAL_TRIGGER_SELECTION
* config->triggerSource = kTPM_TriggerSource_External;
#endif
*
```

### Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>config</i> | Pointer to user's TPM config structure. |
|---------------|-----------------------------------------|

### 38.5.4 **status\_t** TPM\_SetupPwm ( **TPM\_Type** \* *base*, **const tpm\_chnl\_pwm\_signal\_param\_t** \* *chnlParams*, **uint8\_t** *numOfChnls*, **tpm\_pwm\_mode\_t** *mode*, **uint32\_t** *pwmFreq\_Hz*, **uint32\_t** *srcClock\_Hz* )

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

## Parameters

|                    |                                                                                     |
|--------------------|-------------------------------------------------------------------------------------|
| <i>base</i>        | TPM peripheral base address                                                         |
| <i>chnlParams</i>  | Array of PWM channel parameters to configure the channel(s)                         |
| <i>numOfChnls</i>  | Number of channels to configure, this should be the size of the array passed in     |
| <i>mode</i>        | PWM operation mode, options available in enumeration <a href="#">tpm_pwm_mode_t</a> |
| <i>pwmFreq_Hz</i>  | PWM signal frequency in Hz                                                          |
| <i>srcClock_Hz</i> | TPM counter clock in Hz                                                             |

## Returns

kStatus\_Success if the PWM setup was successful, kStatus\_Error on failure

**38.5.5 void TPM\_UpdatePwmDutycycle ( **TPM\_Type** \* *base*, **tpm\_chnl\_t** *chnlNumber*, **tpm\_pwm\_mode\_t** *currentPwmMode*, **uint8\_t** *dutyCyclePercent* )**

## Parameters

|                         |                                                                                                                               |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>             | TPM peripheral base address                                                                                                   |
| <i>chnlNumber</i>       | The channel number. In combined mode, this represents the channel pair number                                                 |
| <i>currentPwmMode</i>   | The current PWM mode set during PWM setup                                                                                     |
| <i>dutyCyclePercent</i> | New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle) |

**38.5.6 void TPM\_UpdateChnlEdgeLevelSelect ( **TPM\_Type** \* *base*, **tpm\_chnl\_t** *chnlNumber*, **uint8\_t** *level* )**

## Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

## Function Documentation

|                   |                                                                                                                                                       |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>chnlNumber</i> | The channel number                                                                                                                                    |
| <i>level</i>      | The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field. |

### 38.5.7 void TPM\_SetupInputCapture ( **TPM\_Type** \* *base*, **tpm\_chnl\_t** *chnlNumber*, **tpm\_input\_capture\_edge\_t** *captureMode* )

When the edge specified in the captureMode argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

|                    |                                 |
|--------------------|---------------------------------|
| <i>base</i>        | TPM peripheral base address     |
| <i>chnlNumber</i>  | The channel number              |
| <i>captureMode</i> | Specifies which edge to capture |

### 38.5.8 void TPM\_SetupOutputCompare ( **TPM\_Type** \* *base*, **tpm\_chnl\_t** *chnlNumber*, **tpm\_output\_compare\_mode\_t** *compareMode*, **uint32\_t** *compareValue* )

When the TPM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| <i>base</i>         | TPM peripheral base address                                            |
| <i>chnlNumber</i>   | The channel number                                                     |
| <i>compareMode</i>  | Action to take on the channel output when the compare condition is met |
| <i>compareValue</i> | Value to be programmed in the CnV register.                            |

### 38.5.9 void TPM\_SetupDualEdgeCapture ( **TPM\_Type** \* *base*, **tpm\_chnl\_t** *chnlPairNumber*, **const tpm\_dual\_edge\_capture\_param\_t** \* *edgeParam*, **uint32\_t** *filterValue* )

This function allows to measure a pulse width of the signal on the input of channel of a channel pair. The filter function is disabled if the filterVal argument passed is zero.

Parameters

|                        |                                                     |
|------------------------|-----------------------------------------------------|
| <i>base</i>            | TPM peripheral base address                         |
| <i>chnlPair-Number</i> | The TPM 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.          |

**38.5.10 void TPM\_SetupQuadDecode ( TPM\_Type \* *base*, const tpm\_phase\_params\_t \* *phaseAParams*, const tpm\_phase\_params\_t \* *phaseBParams*, tpm\_quad\_decode\_mode\_t *quadMode* )**

Parameters

|                     |                                                       |
|---------------------|-------------------------------------------------------|
| <i>base</i>         | TPM 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 |

**38.5.11 void TPM\_EnableInterrupts ( TPM\_Type \* *base*, uint32\_t *mask* )**

Parameters

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | TPM peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">tpm_interrupt_enable_t</a> |

**38.5.12 void TPM\_DisableInterrupts ( TPM\_Type \* *base*, uint32\_t *mask* )**

Parameters

## Function Documentation

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | TPM peripheral base address                                                                                          |
| <i>mask</i> | The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">tpm_interrupt_enable_t</a> |

### 38.5.13 **uint32\_t TPM\_GetEnabledInterrupts ( TPM\_Type \* *base* )**

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [tpm\\_interrupt\\_enable\\_t](#)

### 38.5.14 **static uint32\_t TPM\_GetStatusFlags ( TPM\_Type \* *base* ) [inline], [static]**

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

Returns

The status flags. This is the logical OR of members of the enumeration [tpm\\_status\\_flags\\_t](#)

### 38.5.15 **static void TPM\_ClearStatusFlags ( TPM\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

|             |                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------|
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">tpm_status_flags_t</a> |
|-------------|------------------------------------------------------------------------------------------------------------------|

### 38.5.16 static void TPM\_StartTimer ( **TPM\_Type** \* *base*, **tpm\_clock\_source\_t** *clockSource* ) [inline], [static]

Parameters

|                    |                                                                           |
|--------------------|---------------------------------------------------------------------------|
| <i>base</i>        | TPM peripheral base address                                               |
| <i>clockSource</i> | TPM clock source; once clock source is set the counter will start running |

### 38.5.17 static void TPM\_StopTimer ( **TPM\_Type** \* *base* ) [inline], [static]

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

## Function Documentation

## Chapter 39

# UART: Universal Asynchronous Receiver/Transmitter Driver

### 39.1 Overview

#### Modules

- [UART DMA Driver](#)
- [UART Driver](#)
- [UART FreeRTOS Driver](#)
- [UART eDMA Driver](#)
- [UART μCOS/II Driver](#)
- [UART μCOS/III Driver](#)

### 39.2 UART Driver

#### 39.2.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of Kinetis devices.

The UART driver includes functional APIs and transactional APIs.

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

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `uart_handle_t` as the second parameter. Initialize the handle by calling the [UART\\_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer, which means that the functions [UART\\_TransferSendNonBlocking\(\)](#) and [UART\\_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_UART_TxIdle` and `kStatus_UART_RxIdle`.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the [UART\\_TransferCreateHandle\(\)](#). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The [UART\\_TransferReceiveNonBlocking\(\)](#) function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the `kStatus_UART_RxIdle`.

If the receive ring buffer is full, the upper layer is informed through a callback with the `kStatus_UART_RxRingBufferOverrun`. In the callback function, the upper layer reads data out from the ring buffer. If not, existing data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code.

```
UART_TransferCreateHandle(UART0, &handle, UART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

#### 39.2.2 Typical use case

##### 39.2.2.1 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);
}

```

### 39.2.2.2 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)
{
}

// ...
}
```

### 39.2.2.3 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)
 {
 }
 }

 // ...
}

```

### 39.2.2.4 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\_RxRingBufferOverrun = MAKE\_STATUS(kStatusGroup\_UART, 8),
   
kStatus\_UART\_RxHardwareOverrun = MAKE\_STATUS(kStatusGroup\_UART, 9),
   
kStatus\_UART\_NoiseError = MAKE\_STATUS(kStatusGroup\_UART, 10),
   
kStatus\_UART\_FramingError = MAKE\_STATUS(kStatusGroup\_UART, 11),
   
kStatus\_UART\_ParityError = MAKE\_STATUS(kStatusGroup\_UART, 12),
   
kStatus\_UART\_BaudrateNotSupport }
   
*Error codes for the UART driver.*
- enum `uart_parity_mode_t` {
   
kUART\_ParityDisabled = 0x0U,
   
kUART\_ParityEven = 0x2U,
   
kUART\_ParityOdd = 0x3U }
   
*UART parity mode.*
- enum `uart_stop_bit_count_t` {
   
kUART\_OneStopBit = 0U,
   
kUART\_TwoStopBit = 1U }
   
*UART stop bit count.*
- enum `_uart_interrupt_enable` {
   
kUART\_LinBreakInterruptEnable = (UART\_BDH\_LBKDIIE\_MASK),
   
kUART\_RxActiveEdgeInterruptEnable = (UART\_BDH\_RXEDGIE\_MASK),
   
kUART\_TxDataRegEmptyInterruptEnable = (UART\_C2\_TIE\_MASK << 8),
   
kUART\_TransmissionCompleteInterruptEnable = (UART\_C2\_TCIE\_MASK << 8),
   
kUART\_RxDataRegFullInterruptEnable = (UART\_C2\_RIE\_MASK << 8),
   
kUART\_IdleLineInterruptEnable = (UART\_C2\_ILIE\_MASK << 8),
   
kUART\_RxOverrunInterruptEnable = (UART\_C3\_ORIE\_MASK << 16),
   
kUART\_NoiseErrorInterruptEnable = (UART\_C3\_NEIE\_MASK << 16),
   
kUART\_FramingErrorInterruptEnable = (UART\_C3\_FEIE\_MASK << 16),
   
kUART\_ParityErrorInterruptEnable = (UART\_C3\_PEIE\_MASK << 16),
   
kUART\_RxFifoOverflowInterruptEnable = (UART\_CFIFO\_RXOFE\_MASK << 24),
   
kUART\_TxFifoOverflowInterruptEnable = (UART\_CFIFO\_TXOFE\_MASK << 24),
   
kUART\_RxFifoUnderflowInterruptEnable = (UART\_CFIFO\_RXUFE\_MASK << 24) }
   
*UART interrupt configuration structure, default settings all disabled.*
- enum `_uart_flags` {

## UART Driver

```
kUART_TxDataRegEmptyFlag = (UART_S1_TDRE_MASK),
kUART_TransmissionCompleteFlag = (UART_S1_TC_MASK),
kUART_RxDataRegFullFlag = (UART_S1_RDRF_MASK),
kUART_IdleLineFlag = (UART_S1_IDLE_MASK),
kUART_RxOverrunFlag = (UART_S1_OR_MASK),
kUART_NoiseErrorFlag = (UART_S1_NF_MASK),
kUART_FramingErrorFlag = (UART_S1_FE_MASK),
kUART_ParityErrorFlag = (UART_S1_PF_MASK),
kUART_LinBreakFlag,
kUART_RxActiveEdgeFlag,
kUART_RxActiveFlag,
kUART_NoiseErrorInRxDataRegFlag = (UART_ED_NOISY_MASK << 16),
kUART_ParityErrorInRxDataRegFlag = (UART_ED_PARITYE_MASK << 16),
kUART_TxFifoEmptyFlag = (UART_SFIFO_TXEMPT_MASK << 24),
kUART_RxFifoEmptyFlag = (UART_SFIFO_RXEMPT_MASK << 24),
kUART_TxFifoOverflowFlag = (UART_SFIFO_TXOF_MASK << 24),
kUART_RxFifoOverflowFlag = (UART_SFIFO_RXOF_MASK << 24),
kUART_RxFifoUnderflowFlag = (UART_SFIFO_RXUF_MASK << 24) }
```

*UART status flags.*

## Driver version

- #define **FSL\_UART\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 4))  
*UART driver version 2.1.4.*

## Initialization and deinitialization

- status\_t **UART\_Init** (UART\_Type \*base, const **uart\_config\_t** \*config, uint32\_t srcClock\_Hz)  
*Initializes a UART instance with a user configuration structure and peripheral clock.*
- void **UART\_Deinit** (UART\_Type \*base)  
*Deinitializes a UART instance.*
- void **UART\_GetDefaultConfig** (**uart\_config\_t** \*config)  
*Gets the default configuration structure.*
- status\_t **UART\_SetBaudRate** (UART\_Type \*base, uint32\_t baudRate\_Bps, uint32\_t srcClock\_Hz)  
*Sets the UART instance baud rate.*

## Status

- uint32\_t **UART\_GetStatusFlags** (UART\_Type \*base)  
*Gets UART status flags.*
- status\_t **UART\_ClearStatusFlags** (UART\_Type \*base, uint32\_t mask)  
*Clears status flags with the provided mask.*

## Interrupts

- void **UART\_EnableInterrupts** (UART\_Type \*base, uint32\_t mask)  
*Enables UART interrupts according to the provided mask.*
- void **UART\_DisableInterrupts** (UART\_Type \*base, uint32\_t mask)  
*Disables the UART interrupts according to the provided mask.*
- uint32\_t **UART\_GetEnabledInterrupts** (UART\_Type \*base)  
*Gets the enabled UART interrupts.*

## DMA Control

- static uint32\_t **UART\_GetDataRegisterAddress** (UART\_Type \*base)  
*Gets the UART data register address.*
- static void **UART\_EnableTxDMA** (UART\_Type \*base, bool enable)  
*Enables or disables the UART transmitter DMA request.*
- static void **UART\_EnableRxDMA** (UART\_Type \*base, bool enable)  
*Enables or disables the UART receiver DMA.*

## Bus Operations

- static void **UART\_EnableTx** (UART\_Type \*base, bool enable)  
*Enables or disables the UART transmitter.*
- static void **UART\_EnableRx** (UART\_Type \*base, bool enable)  
*Enables or disables the UART receiver.*
- static void **UART\_WriteByte** (UART\_Type \*base, uint8\_t data)  
*Writes to the TX register.*
- static uint8\_t **UART\_ReadByte** (UART\_Type \*base)  
*Reads the RX register directly.*
- void **UART\_WriteBlocking** (UART\_Type \*base, const uint8\_t \*data, size\_t length)  
*Writes to the TX register using a blocking method.*
- status\_t **UART\_ReadBlocking** (UART\_Type \*base, uint8\_t \*data, size\_t length)  
*Read RX data register using a blocking method.*

## Transactional

- void **UART\_TransferCreateHandle** (UART\_Type \*base, uart\_handle\_t \*handle, **uart\_transfer\_callback\_t** callback, void \*userData)  
*Initializes the UART handle.*
- void **UART\_TransferStartRingBuffer** (UART\_Type \*base, uart\_handle\_t \*handle, uint8\_t \*ringBuffer, size\_t ringBufferSize)  
*Sets up the RX ring buffer.*
- void **UART\_TransferStopRingBuffer** (UART\_Type \*base, uart\_handle\_t \*handle)  
*Aborts the background transfer and uninstalls the ring buffer.*
- status\_t **UART\_TransferSendNonBlocking** (UART\_Type \*base, uart\_handle\_t \*handle, **uart\_transfer\_t** \*xfer)  
*Transmits a buffer of data using the interrupt method.*

## UART Driver

- 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.*

### 39.2.3 Data Structure Documentation

#### 39.2.3.1 struct uart\_config\_t

##### Data Fields

- uint32\_t baudRate\_Bps  
*UART baud rate.*
- [uart\\_parity\\_mode\\_t](#) parityMode  
*Parity mode, disabled (default), even, odd.*
- [uart\\_stop\\_bit\\_count\\_t](#) stopBitCount  
*Number of stop bits, 1 stop bit (default) or 2 stop bits.*
- uint8\_t txFifoWatermark  
*TX FIFO watermark.*
- uint8\_t rxFifoWatermark  
*RX FIFO watermark.*
- bool enableTx  
*Enable TX.*
- bool enableRx  
*Enable RX.*

#### 39.2.3.2 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.*

### 39.2.3.2.0.93 Field Documentation

#### 39.2.3.2.0.93.1 `uint8_t* uart_transfer_t::data`

#### 39.2.3.2.0.93.2 `size_t uart_transfer_t::dataSize`

### 39.2.3.3 `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

### 39.2.3.3.0.94 Field Documentation

- 39.2.3.3.0.94.1 `uint8_t* volatile uart_handle_t::txData`
- 39.2.3.3.0.94.2 `volatile size_t uart_handle_t::txDataSize`
- 39.2.3.3.0.94.3 `size_t uart_handle_t::txDataSizeAll`
- 39.2.3.3.0.94.4 `uint8_t* volatile uart_handle_t::rxData`
- 39.2.3.3.0.94.5 `volatile size_t uart_handle_t::rxDataSize`
- 39.2.3.3.0.94.6 `size_t uart_handle_t::rxDataSizeAll`
- 39.2.3.3.0.94.7 `uint8_t* uart_handle_t::rxRingBuffer`
- 39.2.3.3.0.94.8 `size_t uart_handle_t::rxRingBufferSize`
- 39.2.3.3.0.94.9 `volatile uint16_t uart_handle_t::rxRingBufferHead`
- 39.2.3.3.0.94.10 `volatile uint16_t uart_handle_t::rxRingBufferTail`
- 39.2.3.3.0.94.11 `uart_transfer_callback_t uart_handle_t::callback`
- 39.2.3.3.0.94.12 `void* uart_handle_t::userData`
- 39.2.3.3.0.94.13 `volatile uint8_t uart_handle_t::txState`

### 39.2.4 Macro Definition Documentation

- 39.2.4.1 `#define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))`

### 39.2.5 Typedef Documentation

- 39.2.5.1 `typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)`

### 39.2.6 Enumeration Type Documentation

#### 39.2.6.1 enum \_uart\_status

Enumerator

- `kStatus_UART_TxBusy` Transmitter is busy.
- `kStatus_UART_RxBusy` Receiver is busy.
- `kStatus_UART_TxIdle` UART transmitter is idle.
- `kStatus_UART_RxIdle` UART receiver is idle.
- `kStatus_UART_TxWatermarkTooLarge` TX FIFO watermark too large.

***kStatus\_UART\_RxWatermarkTooLarge*** RX FIFO watermark too large.  
***kStatus\_UART\_FlagCannotClearManually*** UART flag can't be manually cleared.  
***kStatus\_UART\_Error*** Error happens on UART.  
***kStatus\_UART\_RxRingBufferOverrun*** UART RX software ring buffer overrun.  
***kStatus\_UART\_RxHardwareOverrun*** UART RX receiver overrun.  
***kStatus\_UART\_NoiseError*** UART noise error.  
***kStatus\_UART\_FramingError*** UART framing error.  
***kStatus\_UART\_ParityError*** UART parity error.  
***kStatus\_UART\_BaudrateNotSupport*** Baudrate is not support in current clock source.

### 39.2.6.2 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.

### 39.2.6.3 enum uart\_stop\_bit\_count\_t

Enumerator

***kUART\_OneStopBit*** One stop bit.  
***kUART\_TwoStopBit*** Two stop bits.

### 39.2.6.4 enum \_uart\_interrupt\_enable

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

Enumerator

***kUART\_LinBreakInterruptEnable*** LIN break detect interrupt.  
***kUART\_RxActiveEdgeInterruptEnable*** RX active edge interrupt.  
***kUART\_TxDataRegEmptyInterruptEnable*** Transmit data register empty interrupt.  
***kUART\_TransmissionCompleteInterruptEnable*** Transmission complete interrupt.  
***kUART\_RxDataRegFullInterruptEnable*** Receiver data register full interrupt.  
***kUART\_IdleLineInterruptEnable*** Idle line interrupt.  
***kUART\_RxOverrunInterruptEnable*** Receiver overrun interrupt.  
***kUART\_NoiseErrorInterruptEnable*** Noise error flag interrupt.  
***kUART\_FramingErrorInterruptEnable*** Framing error flag interrupt.  
***kUART\_ParityErrorInterruptEnable*** Parity error flag interrupt.  
***kUART\_RxFifoOverflowInterruptEnable*** RX FIFO overflow interrupt.  
***kUART\_TxFifoOverflowInterruptEnable*** TX FIFO overflow interrupt.  
***kUART\_RxFifoUnderflowInterruptEnable*** RX FIFO underflow interrupt.

### 39.2.6.5 enum \_uart\_flags

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

Enumerator

- kUART\_TxDataRegEmptyFlag* TX data register empty flag.
- kUART\_TransmissionCompleteFlag* Transmission complete flag.
- kUART\_RxDataRegFullFlag* RX data register full flag.
- kUART\_IdleLineFlag* Idle line detect flag.
- kUART\_RxOverrunFlag* RX overrun flag.
- kUART\_NoiseErrorFlag* RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets
- kUART\_FramingErrorFlag* Frame error flag, sets if logic 0 was detected where stop bit expected.
- kUART\_ParityErrorFlag* If parity enabled, sets upon parity error detection.
- kUART\_LinBreakFlag* LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.
- kUART\_RxActiveEdgeFlag* RX pin active edge interrupt flag, sets when active edge detected.
- kUART\_RxActiveFlag* Receiver Active Flag (RAF), sets at beginning of valid start bit.
- kUART\_NoiseErrorInRxDataRegFlag* Noisy bit, sets if noise detected.
- kUART\_ParityErrorInRxDataRegFlag* Parity bit, sets if parity error detected.
- kUART\_TxFifoEmptyFlag* TXEMPT bit, sets if TX buffer is empty.
- kUART\_RxFifoEmptyFlag* RXEMPT bit, sets if RX buffer is empty.
- kUART\_TxFifoOverflowFlag* TXOF bit, sets if TX buffer overflow occurred.
- kUART\_RxFifoOverflowFlag* RXOF bit, sets if receive buffer overflow.
- kUART\_RxFifoUnderflowFlag* RXUF bit, sets if receive buffer underflow.

### 39.2.7 Function Documentation

#### 39.2.7.1 status\_t **UART\_Init** ( **UART\_Type** \* *base*, **const uart\_config\_t** \* *config*, **uint32\_t** *srcClock\_Hz* )

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

```
* uart_config_t uartConfig;
* uartConfig.baudRate_Bps = 115200U;
* uartConfig.parityMode = kUART_ParityDisabled;
* uartConfig.stopBitCount = kUART_OneStopBit;
* uartConfig.txFifoWatermark = 0;
* uartConfig.rxFifoWatermark = 1;
* UART_Init(UART1, &uartConfig, 20000000U);
*
```

Parameters

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

Return values

|                                         |                                                  |
|-----------------------------------------|--------------------------------------------------|
| <i>kStatus_UART_Baudrate-NotSupport</i> | Baudrate is not support in current clock source. |
| <i>kStatus_Success</i>                  | Status UART initialize succeed                   |

### 39.2.7.2 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. |
|-------------|-------------------------------|

### 39.2.7.3 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. |
|---------------|-------------------------------------|

### 39.2.7.4 status\_t **UART\_SetBaudRate** ( **UART\_Type** \* *base*, **uint32\_t** *baudRate\_Bps*, **uint32\_t** *srcClock\_Hz* )

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the **UART\_Init**.

```
* UART_SetBaudRate(UART1, 115200U, 20000000U);
*
```

## UART Driver

Parameters

|                     |                                    |
|---------------------|------------------------------------|
| <i>base</i>         | UART peripheral base address.      |
| <i>baudRate_Bps</i> | UART baudrate to be set.           |
| <i>srcClock_Hz</i>  | UART clock source frequency in Hz. |

Return values

|                                         |                                                      |
|-----------------------------------------|------------------------------------------------------|
| <i>kStatus_UART_Baudrate-NotSupport</i> | Baudrate is not support in the current clock source. |
| <i>kStatus_Success</i>                  | Set baudrate succeeded.                              |

### 39.2.7.5 `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`.

### 39.2.7.6 `status_t UART_ClearStatusFlags ( UART_Type * base, uint32_t mask )`

This function clears UART status flags with a provided mask. An automatically cleared flag can't be cleared by this function. These flags can only be cleared or set by hardware. kUART\_TxDataRegEmptyFlag, kUART\_TransmissionCompleteFlag, kUART\_RxDataRegFullFlag, kUART\_RxActiveFlag, kUART\_NoiseErrorInRxDataRegFlag, kUART\_ParityErrorInRxDataRegFlag, kUART\_TxFifoEmptyFlag, kUART\_RxFifoEmptyFlag Note that this API should be called when the Tx/Rx is idle. Otherwise it has no effect.

## Parameters

|             |                                                                                         |
|-------------|-----------------------------------------------------------------------------------------|
| <i>base</i> | UART peripheral base address.                                                           |
| <i>mask</i> | The status flags to be cleared; it is logical OR value of <a href="#">_uart_flags</a> . |

## Return values

|                                             |                                                                                         |
|---------------------------------------------|-----------------------------------------------------------------------------------------|
| <i>kStatus_UART_FlagCannotClearManually</i> | The flag can't be cleared by this function but it is cleared automatically by hardware. |
| <i>kStatus_Success</i>                      | Status in the mask is cleared.                                                          |

**39.2.7.7 void UART\_EnableInterrupts ( **UART\_Type** \* *base*, **uint32\_t** *mask* )**

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [\\_uart\\_interrupt\\_enable](#). For example, to enable TX empty interrupt and RX full interrupt, do the following.

```
* UART_EnableInterrupts(UART1,
 kUART_TxDataRegEmptyInterruptEnable |
 kUART_RxDataRegFullInterruptEnable);
*
```

## Parameters

|             |                                                                                  |
|-------------|----------------------------------------------------------------------------------|
| <i>base</i> | UART peripheral base address.                                                    |
| <i>mask</i> | The interrupts to enable. Logical OR of <a href="#">_uart_interrupt_enable</a> . |

**39.2.7.8 void UART\_DisableInterrupts ( **UART\_Type** \* *base*, **uint32\_t** *mask* )**

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [\\_uart\\_interrupt\\_enable](#). For example, to disable TX empty interrupt and RX full interrupt do the following.

```
* UART_DisableInterrupts(UART1,
 kUART_TxDataRegEmptyInterruptEnable |
 kUART_RxDataRegFullInterruptEnable);
*
```

## UART Driver

Parameters

|             |                                                                                   |
|-------------|-----------------------------------------------------------------------------------|
| <i>base</i> | UART peripheral base address.                                                     |
| <i>mask</i> | The interrupts to disable. Logical OR of <a href="#">_uart_interrupt_enable</a> . |

### 39.2.7.9 **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](#).

### 39.2.7.10 **static uint32\_t UART\_GetDataRegisterAddress ( *UART\_Type* \* *base* ) [inline], [static]**

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

Parameters

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

Returns

UART data register addresses which are used both by the transmitter and the receiver.

### 39.2.7.11 **static void UART\_EnableTxDMA ( *UART\_Type* \* *base*, *bool enable* ) [inline], [static]**

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

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | UART peripheral base address.     |
| <i>enable</i> | True to enable, false to disable. |

### 39.2.7.12 static void UART\_EnableRxDMA ( **UART\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function enables or disables the receiver data register full flag, S1[RDRF], to generate DMA requests.

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | UART peripheral base address.     |
| <i>enable</i> | True to enable, false to disable. |

### 39.2.7.13 static void UART\_EnableTx ( **UART\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function enables or disables the UART transmitter.

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | UART peripheral base address.     |
| <i>enable</i> | True to enable, false to disable. |

### 39.2.7.14 static void UART\_EnableRx ( **UART\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function enables or disables the UART receiver.

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | UART peripheral base address.     |
| <i>enable</i> | True to enable, false to disable. |

### 39.2.7.15 static void UART\_WriteByte ( **UART\_Type** \* *base*, **uint8\_t** *data* ) [**inline**], [**static**]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

## UART Driver

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | UART peripheral base address. |
| <i>data</i> | The byte to write.            |

### 39.2.7.16 static uint8\_t UART\_ReadByte ( **UART\_Type** \* *base* ) [inline], [static]

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

Parameters

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

Returns

The byte read from UART data register.

### 39.2.7.17 void UART\_WriteBlocking ( **UART\_Type** \* *base*, **const uint8\_t** \* *data*, **size\_t** *length* )

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Note

This function does not check whether all data is sent out to the bus. Before disabling the TX, check kUART\_TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

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

### 39.2.7.18 status\_t UART\_ReadBlocking ( **UART\_Type** \* *base*, **uint8\_t** \* *data*, **size\_t** *length* )

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

Parameters

|               |                                                         |
|---------------|---------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                           |
| <i>data</i>   | Start address of the buffer to store the received data. |
| <i>length</i> | Size of the buffer.                                     |

Return values

|                                        |                                                 |
|----------------------------------------|-------------------------------------------------|
| <i>kStatus_UART_Rx-HardwareOverrun</i> | Receiver overrun occurred while receiving data. |
| <i>kStatus_UART_Noise-Error</i>        | A noise error occurred while receiving data.    |
| <i>kStatus_UART_Framing-Error</i>      | A framing error occurred while receiving data.  |
| <i>kStatus_UART_Parity-Error</i>       | A parity error occurred while receiving data.   |
| <i>kStatus_Success</i>                 | Successfully received all data.                 |

### 39.2.7.19 void **UART\_TransferCreateHandle** ( **UART\_Type \* base**, **uart\_handle\_t \* handle**, **uart\_transfer\_callback\_t callback**, **void \* userData** )

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

Parameters

|                 |                                         |
|-----------------|-----------------------------------------|
| <i>base</i>     | UART peripheral base address.           |
| <i>handle</i>   | UART handle pointer.                    |
| <i>callback</i> | The callback function.                  |
| <i>userData</i> | The parameter of the callback function. |

### 39.2.7.20 void **UART\_TransferStartRingBuffer** ( **UART\_Type \* base**, **uart\_handle\_t \* handle**, **uint8\_t \* ringBuffer**, **size\_t ringBufferSize** )

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

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

## UART Driver

### Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

### Parameters

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

### 39.2.7.21 void `UART_TransferStopRingBuffer` ( `UART_Type * base, uart_handle_t * handle` )

This function aborts the background transfer and uninstalls the ring buffer.

### Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |

### 39.2.7.22 status\_t `UART_TransferSendNonBlocking` ( `UART_Type * base, uart_handle_t * handle, uart_transfer_t * xfer` )

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the ISR, the UART driver calls the callback function and passes the `kStatus_UART_TxIdle` as status parameter.

### Note

The `kStatus_UART_TxIdle` is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out. Before disabling the TX, check the `kUART_TxTransmissionCompleteFlag` to ensure that the TX is finished.

Parameters

|               |                                                                |
|---------------|----------------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                                  |
| <i>handle</i> | UART handle pointer.                                           |
| <i>xfer</i>   | UART transfer structure. See <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                                                                    |
|--------------------------------|------------------------------------------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully start the data transmission.                                          |
| <i>kStatus_UART_TxBusy</i>     | Previous transmission still not finished; data not all written to TX register yet. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.                                                                  |

### 39.2.7.23 void **UART\_TransferAbortSend** ( **UART\_Type** \* *base*, **uart\_handle\_t** \* *handle* )

This function aborts the interrupt-driven data sending. The user can get the remainBytes to find out how many bytes are not sent out.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |

### 39.2.7.24 **status\_t** **UART\_TransferGetSendCount** ( **UART\_Type** \* *base*, **uart\_handle\_t** \* *handle*, **uint32\_t** \* *count* )

This function gets the number of bytes written to the UART TX register by using the interrupt method.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |
| <i>count</i>  | Send bytes count.             |

Return values

## UART Driver

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

### 39.2.7.25 `status_t UART_TransferReceiveNonBlocking ( UART_Type * base, uart_handle_t * handle, uart_transfer_t * xfer, size_t * receivedBytes )`

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter [k-Status\\_UART\\_RxIdle](#). For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the `xfer->data` and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the `xfer->data`. When all data is received, the upper layer is notified.

#### Parameters

|                      |                                                                |
|----------------------|----------------------------------------------------------------|
| <i>base</i>          | UART peripheral base address.                                  |
| <i>handle</i>        | UART handle pointer.                                           |
| <i>xfer</i>          | UART transfer structure, see <a href="#">uart_transfer_t</a> . |
| <i>receivedBytes</i> | Bytes received from the ring buffer directly.                  |

#### Return values

|                                |                                                      |
|--------------------------------|------------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully queue the transfer into transmit queue. |
| <i>kStatus_UART_RxBusy</i>     | Previous receive request is not finished.            |
| <i>kStatus_InvalidArgument</i> | Invalid argument.                                    |

### 39.2.7.26 `void UART_TransferAbortReceive ( UART_Type * base, uart_handle_t * handle )`

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to know how many bytes are not received yet.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |

### 39.2.7.27 **status\_t UART\_TransferGetReceiveCount ( *UART\_Type* \* *base*, *uart\_handle\_t* \* *handle*, *uint32\_t* \* *count* )**

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

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |
| <i>count</i>  | Receive bytes count.          |

Return values

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

### 39.2.7.28 **void UART\_TransferHandleIRQ ( *UART\_Type* \* *base*, *uart\_handle\_t* \* *handle* )**

This function handles the UART transmit and receive IRQ request.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |

### 39.2.7.29 **void UART\_TransferHandleErrorIRQ ( *UART\_Type* \* *base*, *uart\_handle\_t* \* *handle* )**

This function handles the UART error IRQ request.

## UART Driver

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |

## 39.3 UART DMA Driver

### 39.3.1 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.*

### 39.3.2 Data Structure Documentation

#### 39.3.2.1 struct \_uart\_dma\_handle

##### Data Fields

- [UART\\_Type](#) \* [base](#)

## UART DMA Driver

- *UART peripheral base address.*
- `uart_dma_transfer_callback_t callback`  
*Callback function.*
- `void *userData`  
*UART callback function parameter.*
- `size_t rxDataSizeAll`  
*Size of the data to receive.*
- `size_t txDataSizeAll`  
*Size of the data to send out.*
- `dma_handle_t *txDmaHandle`  
*The DMA TX channel used.*
- `dma_handle_t *rxDmaHandle`  
*The DMA RX channel used.*
- `volatile uint8_t txState`  
*TX transfer state.*
- `volatile uint8_t rxState`  
*RX transfer state.*

### 39.3.2.1.0.95 Field Documentation

39.3.2.1.0.95.1 `UART_Type* uart_dma_handle_t::base`

39.3.2.1.0.95.2 `uart_dma_transfer_callback_t uart_dma_handle_t::callback`

39.3.2.1.0.95.3 `void* uart_dma_handle_t::userData`

39.3.2.1.0.95.4 `size_t uart_dma_handle_t::rxDataSizeAll`

39.3.2.1.0.95.5 `size_t uart_dma_handle_t::txDataSizeAll`

39.3.2.1.0.95.6 `dma_handle_t* uart_dma_handle_t::txDmaHandle`

39.3.2.1.0.95.7 `dma_handle_t* uart_dma_handle_t::rxDmaHandle`

39.3.2.1.0.95.8 `volatile uint8_t uart_dma_handle_t::txState`

### 39.3.3 Typedef Documentation

39.3.3.1 `typedef void(* uart_dma_transfer_callback_t)(UART_Type *base,  
uart_dma_handle_t *handle, status_t status, void *userData)`

### 39.3.4 Function Documentation

39.3.4.1 `void UART_TransferCreateHandleDMA ( UART_Type * base, uart_dma_handle_t  
* handle, uart_dma_transfer_callback_t callback, void * userData,  
dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle )`

Parameters

|                    |                                                          |
|--------------------|----------------------------------------------------------|
| <i>base</i>        | UART peripheral base address.                            |
| <i>handle</i>      | Pointer to the <code>uart_dma_handle_t</code> structure. |
| <i>callback</i>    | UART callback, NULL means no callback.                   |
| <i>userData</i>    | User callback function data.                             |
| <i>rxDmaHandle</i> | User requested DMA handle for the RX DMA transfer.       |
| <i>txDmaHandle</i> | User requested DMA handle for the TX DMA transfer.       |

### 39.3.4.2 `status_t UART_TransferSendDMA ( UART_Type * base, uart_dma_handle_t * handle, uart_transfer_t * xfer )`

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

Parameters

|               |                                                                    |
|---------------|--------------------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                                      |
| <i>handle</i> | UART handle pointer.                                               |
| <i>xfer</i>   | UART DMA transfer structure. See <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_TxBusy</i>     | Previous transfer ongoing.      |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

### 39.3.4.3 `status_t UART_TransferReceiveDMA ( UART_Type * base, uart_dma_handle_t * handle, uart_transfer_t * xfer )`

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

Parameters

## UART DMA Driver

|               |                                                                    |
|---------------|--------------------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                                      |
| <i>handle</i> | Pointer to the <code>uart_dma_handle_t</code> structure.           |
| <i>xfer</i>   | UART DMA transfer structure. See <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_RxBusy</i>     | Previous transfer on going.     |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

### 39.3.4.4 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. |

### 39.3.4.5 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. |

### 39.3.4.6 status\_t `UART_TransferGetSendCountDMA` ( `UART_Type * base, uart_dma_handle_t * handle, uint32_t * count` )

This function gets the number of bytes written to UART TX register by DMA.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |
| <i>count</i>  | Send bytes count.             |

Return values

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

### 39.3.4.7 **status\_t UART\_TransferGetReceiveCountDMA ( *UART\_Type \* base, uart\_dma\_handle\_t \* handle, uint32\_t \* count* )**

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

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |
| <i>count</i>  | Receive bytes count.          |

Return values

|                                      |                                                       |
|--------------------------------------|-------------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | No receive in progress.                               |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

### 39.4 UART eDMA Driver

#### 39.4.1 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.*

#### 39.4.2 Data Structure Documentation

##### 39.4.2.1 struct \_uart\_edma\_handle

###### Data Fields

- [uart\\_edma\\_transfer\\_callback\\_t](#)  callback

- *Callback function.*
- `void *userData`  
*UART callback function parameter.*
- `size_t rxDataSizeAll`  
*Size of the data to receive.*
- `size_t txDataSizeAll`  
*Size of the data to send out.*
- `edma_handle_t *txEdmaHandle`  
*The eDMA TX channel used.*
- `edma_handle_t *rxEdmaHandle`  
*The eDMA RX channel used.*
- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `volatile uint8_t txState`  
*TX transfer state.*
- `volatile uint8_t rxState`  
*RX transfer state.*

### 39.4.2.1.0.96 Field Documentation

39.4.2.1.0.96.1 `uart_edma_transfer_callback_t uart_edma_handle_t::callback`

39.4.2.1.0.96.2 `void* uart_edma_handle_t::userData`

39.4.2.1.0.96.3 `size_t uart_edma_handle_t::rxDataSizeAll`

39.4.2.1.0.96.4 `size_t uart_edma_handle_t::txDataSizeAll`

39.4.2.1.0.96.5 `edma_handle_t* uart_edma_handle_t::txEdmaHandle`

39.4.2.1.0.96.6 `edma_handle_t* uart_edma_handle_t::rxEdmaHandle`

39.4.2.1.0.96.7 `uint8_t uart_edma_handle_t::nbytes`

39.4.2.1.0.96.8 `volatile uint8_t uart_edma_handle_t::txState`

### 39.4.3 Typedef Documentation

39.4.3.1 `typedef void(* uart_edma_transfer_callback_t)(UART_Type *base,  
uart_edma_handle_t *handle, status_t status, void *userData)`

### 39.4.4 Function Documentation

39.4.4.1 `void UART_TransferCreateHandleEDMA ( UART_Type * base,  
uart_edma_handle_t * handle, uart_edma_transfer_callback_t callback, void *  
userData, edma_handle_t * txEdmaHandle, edma_handle_t * rxEdmaHandle )`

## UART eDMA Driver

Parameters

|                     |                                                           |
|---------------------|-----------------------------------------------------------|
| <i>base</i>         | UART peripheral base address.                             |
| <i>handle</i>       | Pointer to the <code>uart_edma_handle_t</code> structure. |
| <i>callback</i>     | UART callback, NULL means no callback.                    |
| <i>userData</i>     | User callback function data.                              |
| <i>rxEdmaHandle</i> | User-requested DMA handle for RX DMA transfer.            |
| <i>txEdmaHandle</i> | User-requested DMA handle for TX DMA transfer.            |

### 39.4.4.2 `status_t UART_SendEDMA ( UART_Type * base, uart_edma_handle_t * handle, uart_transfer_t * xfer )`

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

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                                       |
| <i>handle</i> | UART handle pointer.                                                |
| <i>xfer</i>   | UART eDMA transfer structure. See <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_TxBusy</i>     | Previous transfer ongoing.      |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

### 39.4.4.3 `status_t UART_ReceiveEDMA ( UART_Type * base, uart_edma_handle_t * handle, uart_transfer_t * xfer )`

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

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                                       |
| <i>handle</i> | Pointer to the <code>uart_edma_handle_t</code> structure.           |
| <i>xfer</i>   | UART eDMA transfer structure. See <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_RxBusy</i>     | Previous transfer ongoing.      |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

#### **39.4.4.4 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. |

#### **39.4.4.5 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. |

#### **39.4.4.6 status\_t `UART_TransferGetSendCountEDMA` ( `UART_Type * base, uart_edma_handle_t * handle, uint32_t * count` )**

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

## UART eDMA Driver

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |
| <i>count</i>  | Send bytes count.             |

Return values

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

### 39.4.4.7 **status\_t UART\_TransferGetReceiveCountEDMA ( *UART\_Type \* base*, *uart\_edma\_handle\_t \* handle*, *uint32\_t \* count* )**

This function gets the number of received bytes.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | UART peripheral base address. |
| <i>handle</i> | UART handle pointer.          |
| <i>count</i>  | Receive bytes count.          |

Return values

|                                      |                                                       |
|--------------------------------------|-------------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | No receive in progress.                               |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

## 39.5 UART FreeRTOS Driver

### 39.5.1 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.*

### 39.5.2 Data Structure Documentation

#### 39.5.2.1 struct `uart_rtos_config_t`

UART RTOS configuration structure.

#### Data Fields

- `UART_Type * base`  
*UART base address.*
- `uint32_t srclk`  
*UART source clock in Hz.*
- `uint32_t baudrate`  
*Desired communication speed.*
- `uart_parity_mode_t parity`  
*Parity setting.*
- `uart_stop_bit_count_t stopbits`  
*Number of stop bits to use.*
- `uint8_t * buffer`  
*Buffer for background reception.*

## UART FreeRTOS Driver

- `uint32_t buffer_size`  
*Size of buffer for background reception.*

### 39.5.3 Function Documentation

#### 39.5.3.1 int `UART_RTOs_Init` ( `uart_rtos_handle_t * handle`, `uart_handle_t * t_handle`, `const uart_rtos_config_t * cfg` )

Parameters

|                       |                                                                                     |
|-----------------------|-------------------------------------------------------------------------------------|
| <code>handle</code>   | The RTOS UART handle, the pointer to an allocated space for RTOS context.           |
| <code>t_handle</code> | The pointer to the allocated space to store the transactional layer internal state. |
| <code>cfg</code>      | The pointer to the parameters required to configure the UART after initialization.  |

Returns

0 succeed; otherwise fail.

#### 39.5.3.2 int `UART_RTOs_Deinit` ( `uart_rtos_handle_t * handle` )

This function deinitializes the UART module, sets all register values to reset value, and frees the resources.

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

#### 39.5.3.3 int `UART_RTOs_Send` ( `uart_rtos_handle_t * handle`, `const uint8_t * buffer`, `uint32_t length` )

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

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

|               |                                    |
|---------------|------------------------------------|
| <i>buffer</i> | The pointer to the buffer to send. |
| <i>length</i> | The number of bytes to send.       |

### 39.5.3.4 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 <i>size_t</i> where the number of received data is filled. |

### 39.6 UART µCOS/II Driver

#### 39.6.1 Overview

#### Data Structures

- struct `uart_rtos_config_t`  
*UART configuration structure.* [More...](#)

#### UART RTOS Operation

- int `UART_RTOS_Init` (`uart_rtos_handle_t *handle`, `uart_handle_t *t_handle`, `const uart_rtos_config_t *cfg`)  
*Initializes a UART instance for operation in RTOS.*
- int `UART_RTOS_Deinit` (`uart_rtos_handle_t *handle`)  
*Deinitializes a UART instance for operation.*

#### UART transactional Operation

- int `UART_RTOS_Send` (`uart_rtos_handle_t *handle`, `const uint8_t *buffer`, `uint32_t length`)  
*Sends data in the background.*
- int `UART_RTOS_Receive` (`uart_rtos_handle_t *handle`, `uint8_t *buffer`, `uint32_t length`, `size_t *received`)  
*Receives data.*

#### 39.6.2 Data Structure Documentation

##### 39.6.2.1 struct `uart_rtos_config_t`

UART RTOS configuration structure.

#### Data Fields

- `UART_Type * base`  
*UART base address.*
- `uint32_t srclk`  
*UART source clock in Hz.*
- `uint32_t baudrate`  
*Desired communication speed.*
- `uart_parity_mode_t parity`  
*Parity setting.*
- `uart_stop_bit_count_t stopbits`  
*Number of stop bits to use.*
- `uint8_t * buffer`  
*Buffer for background reception.*

- `uint32_t buffer_size`  
*Size of buffer for background reception.*

### 39.6.3 Function Documentation

#### 39.6.3.1 int UART\_RTOS\_Init ( `uart_rtos_handle_t * handle`, `uart_handle_t * t_handle`, `const uart_rtos_config_t * cfg` )

Parameters

|                            |                                                                                           |
|----------------------------|-------------------------------------------------------------------------------------------|
| <code>handle</code>        | The RTOS UART handle; the pointer to an allocated space for RTOS context.                 |
| <code>uart_t_handle</code> | The pointer to the allocated space where to store the transactional layer internal state. |
| <code>cfg</code>           | The pointer to the parameters required to configure the UART after initialization.        |

Returns

0 Succeed; otherwise fail.

#### 39.6.3.2 int UART\_RTOS\_Deinit ( `uart_rtos_handle_t * handle` )

This function deinitializes the UART module, sets all register values to reset value, and frees the resources.

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

#### 39.6.3.3 int UART\_RTOS\_Send ( `uart_rtos_handle_t * handle`, `const uint8_t * buffer`, `uint32_t length` )

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

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

## UART µCOS/II Driver

|               |                                    |
|---------------|------------------------------------|
| <i>buffer</i> | The pointer to the buffer to send. |
| <i>length</i> | The number of bytes to send.       |

### 39.6.3.4 int UART\_RTOS\_Receive ( *uart\_rtos\_handle\_t \* handle*, *uint8\_t \* buffer*, *uint32\_t length*, *size\_t \* received* )

This function receives data from UART. It is a synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

Parameters

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

## 39.7 UART µCOS/III Driver

### 39.7.1 Overview

#### Data Structures

- struct `uart_rtos_config_t`  
*UART configuration structure.* [More...](#)

#### UART RTOS Operation

- int `UART_RTOS_Init` (`uart_rtos_handle_t *handle`, `uart_handle_t *t_handle`, `const uart_rtos_config_t *cfg`)  
*Initializes a UART instance for operation in RTOS.*
- int `UART_RTOS_Deinit` (`uart_rtos_handle_t *handle`)  
*Deinitializes a UART instance for operation.*

#### UART transactional Operation

- int `UART_RTOS_Send` (`uart_rtos_handle_t *handle`, `const uint8_t *buffer`, `uint32_t length`)  
*Sends data in the background.*
- int `UART_RTOS_Receive` (`uart_rtos_handle_t *handle`, `uint8_t *buffer`, `uint32_t length`, `size_t *received`)  
*Receives data.*

### 39.7.2 Data Structure Documentation

#### 39.7.2.1 struct `uart_rtos_config_t`

UART RTOS configuration structure.

#### Data Fields

- `UART_Type * base`  
*UART base address.*
- `uint32_t srclk`  
*UART source clock in Hz.*
- `uint32_t baudrate`  
*Desired communication speed.*
- `uart_parity_mode_t parity`  
*Parity setting.*
- `uart_stop_bit_count_t stopbits`  
*Number of stop bits to use.*
- `uint8_t * buffer`  
*Buffer for background reception.*

## UART µCOS/III Driver

- `uint32_t buffer_size`  
*Size of buffer for background reception.*

### 39.7.3 Function Documentation

#### 39.7.3.1 int `UART_RRTOS_Init` ( `uart_rtos_handle_t * handle`, `uart_handle_t * t_handle`, `const uart_rtos_config_t * cfg` )

Parameters

|                            |                                                                                    |
|----------------------------|------------------------------------------------------------------------------------|
| <code>handle</code>        | The RTOS UART handle; the pointer to allocated space for RTOS context.             |
| <code>uart_t_handle</code> | The pointer to an allocated space to store transactional layer internal state.     |
| <code>cfg</code>           | The pointer to the parameters required to configure the UART after initialization. |

Returns

0 Succeed; otherwise fail.

#### 39.7.3.2 int `UART_RRTOS_Deinit` ( `uart_rtos_handle_t * handle` )

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

#### 39.7.3.3 int `UART_RRTOS_Send` ( `uart_rtos_handle_t * handle`, `const uint8_t * buffer`, `uint32_t length` )

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

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

|               |                                    |
|---------------|------------------------------------|
| <i>buffer</i> | The pointer to the buffer to send. |
| <i>length</i> | The number of bytes to send.       |

### 39.7.3.4 int UART\_RTOS\_Receive ( *uart\_rtos\_handle\_t* \* *handle*, *uint8\_t* \* *buffer*, *uint32\_t* *length*, *size\_t* \* *received* )

This function receives data from UART. It is a synchronous API. If any data is immediately available, it is returned immediately and the number of bytes received.

#### Parameters

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



# Chapter 40

## VREF: Voltage Reference Driver

### 40.1 Overview

The KSDK provides a peripheral driver for the Crossbar Voltage Reference (VREF) block of Kinetis devices.

The Voltage Reference(VREF) supplies an accurate 1.2 V voltage output that can be trimmed in 0.5 mV steps. VREF can be used in applications to provide a reference voltage to external devices and to internal analog peripherals, such as the ADC, DAC, or CMP. The voltage reference has operating modes that provide different levels of supply rejection and power consumption.

To configure the VREF driver, configure `vref_config_t` structure in one of two ways.

1. Use the `VREF_GetDefaultConfig()` function.
2. Set the parameter in the `vref_config_t` structure.

To initialize the VREF driver, call the `VREF_Init()` function and pass a pointer to the `vref_config_t` structure.

To de-initialize the VREF driver, call the `VREF_Deinit()` function.

### 40.2 Typical use case and example

This example shows how to generate a reference voltage by using the VREF module.

```
vref_config_t vrefUserConfig;
VREF_GetDefaultConfig(&vrefUserConfig); /* Gets a default configuration. */
VREF_Init(VREF, &vrefUserConfig); /* Initializes and configures the VREF module */

/* Do something */

VREF_Deinit(VREF); /* De-initializes the VREF module */
```

## Data Structures

- struct `vref_config_t`  
*The description structure for the VREF module. [More...](#)*

## Enumerations

- enum `vref_buffer_mode_t` {  
    `kVREF_ModeBandgapOnly` = 0U,  
    `kVREF_ModeHighPowerBuffer` = 1U,  
    `kVREF_ModeLowPowerBuffer` = 2U }  
*VREF modes.*

## Function Documentation

### Driver version

- #define **FSL\_VREF\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 0))  
*Version 2.1.0.*

### VREF functional operation

- void **VREF\_Init** (VREF\_Type \*base, const vref\_config\_t \*config)  
*Enables the clock gate and configures the VREF module according to the configuration structure.*
- void **VREF\_Deinit** (VREF\_Type \*base)  
*Stops and disables the clock for the VREF module.*
- void **VREF\_GetDefaultConfig** (vref\_config\_t \*config)  
*Initializes the VREF configuration structure.*
- void **VREF\_SetTrimVal** (VREF\_Type \*base, uint8\_t trimValue)  
*Sets a TRIM value for the reference voltage.*
- static uint8\_t **VREF\_GetTrimVal** (VREF\_Type \*base)  
*Reads the value of the TRIM meaning output voltage.*

## 40.3 Data Structure Documentation

### 40.3.1 struct vref\_config\_t

#### Data Fields

- vref\_buffer\_mode\_t bufferMode  
*Buffer mode selection.*

## 40.4 Macro Definition Documentation

### 40.4.1 #define FSL\_VREF\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 0))

## 40.5 Enumeration Type Documentation

### 40.5.1 enum vref\_buffer\_mode\_t

Enumerator

**kVREF\_ModeBandgapOnly** Bandgap on only, for stabilization and startup.

**kVREF\_ModeHighPowerBuffer** High-power buffer mode enabled.

**kVREF\_ModeLowPowerBuffer** Low-power buffer mode enabled.

## 40.6 Function Documentation

### 40.6.1 void VREF\_Init ( VREF\_Type \* *base*, const vref\_config\_t \* *config* )

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up **vref\_config\_t** parameters and how to call the VREF\_Init function by passing in these parameters. This is an example.

```

* vref_config_t vrefConfig;
* vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
* vrefConfig.enableExternalVoltRef = false;
* vrefConfig.enableLowRef = false;
* VREF_Init(VREF, &vrefConfig);
*

```

## Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | VREF peripheral address.                |
| <i>config</i> | Pointer to the configuration structure. |

**40.6.2 void VREF\_Deinit ( VREF\_Type \* *base* )**

This function should be called to shut down the module. This is an example.

```

* vref_config_t vrefUserConfig;
* VREF_Init(VREF);
* VREF_GetDefaultConfig(&vrefUserConfig);
* ...
* VREF_Deinit(VREF);
*

```

## Parameters

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

**40.6.3 void VREF\_GetDefaultConfig ( vref\_config\_t \* *config* )**

This function initializes the VREF configuration structure to default values. This is an example.

```

* vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
* vrefConfig->enableExternalVoltRef = false;
* vrefConfig->enableLowRef = false;
*

```

## Parameters

|               |                                          |
|---------------|------------------------------------------|
| <i>config</i> | Pointer to the initialization structure. |
|---------------|------------------------------------------|

**40.6.4 void VREF\_SetTrimVal ( VREF\_Type \* *base*, uint8\_t *trimValue* )**

This function sets a TRIM value for the reference voltage. Note that the TRIM value maximum is 0x3F.

## Function Documentation

Parameters

|                  |                                                                                        |
|------------------|----------------------------------------------------------------------------------------|
| <i>base</i>      | VREF peripheral address.                                                               |
| <i>trimValue</i> | Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)). |

### 40.6.5 static uint8\_t VREF\_GetTrimVal ( VREF\_Type \* *base* ) [inline], [static]

This function gets the TRIM value from the TRM register.

Parameters

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

Returns

Six-bit value of trim setting.

# Chapter 41

## WDOG: Watchdog Timer Driver

### 41.1 Overview

The KSDK provides a peripheral driver for the Watchdog module (WDOG) of Kinetis devices.

### 41.2 Typical use case

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableWindowMode = true;
config.windowValue = 0x1ffU;
WDOG_Init(wdog_base, &config);
```

## Data Structures

- struct `wdog_work_mode_t`  
*Defines WDOG work mode. [More...](#)*
- struct `wdog_config_t`  
*Describes WDOG configuration structure. [More...](#)*
- struct `wdog_test_config_t`  
*Describes WDOG test mode configuration structure. [More...](#)*

## Enumerations

- enum `wdog_clock_source_t` {  
  kWDOG\_LpoClockSource = 0U,  
  kWDOG\_AlternateClockSource = 1U }  
*Describes WDOG clock source.*
- enum `wdog_clock_prescaler_t` {  
  kWDOG\_ClockPrescalerDivide1 = 0x0U,  
  kWDOG\_ClockPrescalerDivide2 = 0x1U,  
  kWDOG\_ClockPrescalerDivide3 = 0x2U,  
  kWDOG\_ClockPrescalerDivide4 = 0x3U,  
  kWDOG\_ClockPrescalerDivide5 = 0x4U,  
  kWDOG\_ClockPrescalerDivide6 = 0x5U,  
  kWDOG\_ClockPrescalerDivide7 = 0x6U,  
  kWDOG\_ClockPrescalerDivide8 = 0x7U }  
*Describes the selection of the clock prescaler.*
- enum `wdog_test_mode_t` {  
  kWDOG\_QuickTest = 0U,  
  kWDOG\_ByteTest = 1U }  
*Describes WDOG test mode.*

## Typical use case

- enum `wdog_tested_byte_t` {  
  `kWDOG_TestByte0` = 0U,  
  `kWDOG_TestByte1` = 1U,  
  `kWDOG_TestByte2` = 2U,  
  `kWDOG_TestByte3` = 3U }  
    *Describes WDOG tested byte selection in byte test mode.*
- enum `_wdog_interrupt_enable_t` { `kWDOG_InterruptEnable` = `WDOG_STCTRLH_IRQRSTEN_MASK` }  
    *WDOG interrupt configuration structure, default settings all disabled.*
- enum `_wdog_status_flags_t` {  
  `kWDOG_RunningFlag` = `WDOG_STCTRLH_WDOGEN_MASK`,  
  `kWDOG_TimeoutFlag` = `WDOG_STCTRLL_INTFLG_MASK` }  
    *WDOG status flags.*

## Driver version

- `#define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`  
    *Defines WDOG driver version 2.0.0.*

## Unlock sequence

- `#define WDOG_FIRST_WORD_OF_UNLOCK (0xC520U)`  
    *First word of unlock sequence.*
- `#define WDOG_SECOND_WORD_OF_UNLOCK (0xD928U)`  
    *Second word of unlock sequence.*

## Refresh sequence

- `#define WDOG_FIRST_WORD_OF_REFRESH (0xA602U)`  
    *First word of refresh sequence.*
- `#define WDOG_SECOND_WORD_OF_REFRESH (0xB480U)`  
    *Second word of refresh sequence.*

## WDOG Initialization and De-initialization

- `void WDOG_GetDefaultConfig (wdog_config_t *config)`  
    *Initializes the WDOG configuration structure.*
- `void WDOG_Init (WDOG_Type *base, const wdog_config_t *config)`  
    *Initializes the WDOG.*
- `void WDOG_Deinit (WDOG_Type *base)`  
    *Shuts down the WDOG.*
- `void WDOG_SetTestModeConfig (WDOG_Type *base, wdog_test_config_t *config)`  
    *Configures the WDOG functional test.*

## WDOG Functional Operation

- `static void WDOG_Enable (WDOG_Type *base)`  
    *Enables the WDOG module.*
- `static void WDOG_Disable (WDOG_Type *base)`

- static void [WDOG\\_EnableInterrupts](#) (WDOG\_Type \*base, uint32\_t mask)
 

*Disables the WDOG module.*
- static void [WDOG\\_DisableInterrupts](#) (WDOG\_Type \*base, uint32\_t mask)
 

*Enables the WDOG interrupt.*
- uint32\_t [WDOG\\_GetStatusFlags](#) (WDOG\_Type \*base)
 

*Gets the WDOG all status flags.*
- void [WDOG\\_ClearStatusFlags](#) (WDOG\_Type \*base, uint32\_t mask)
 

*Clears the WDOG flag.*
- static void [WDOG\\_SetTimeoutValue](#) (WDOG\_Type \*base, uint32\_t timeoutCount)
 

*Sets the WDOG timeout value.*
- static void [WDOG\\_SetWindowValue](#) (WDOG\_Type \*base, uint32\_t windowValue)
 

*Sets the WDOG window value.*
- static void [WDOG\\_Unlock](#) (WDOG\_Type \*base)
 

*Unlocks the WDOG register written.*
- void [WDOG\\_Refresh](#) (WDOG\_Type \*base)
 

*Refreshes the WDOG timer.*
- static uint16\_t [WDOG\\_GetResetCount](#) (WDOG\_Type \*base)
 

*Gets the WDOG reset count.*
- static void [WDOG\\_ClearResetCount](#) (WDOG\_Type \*base)
 

*Clears the WDOG reset count.*

## 41.3 Data Structure Documentation

### 41.3.1 struct wdog\_work\_mode\_t

#### Data Fields

- bool [enableWait](#)

*Enables or disables WDOG in wait mode.*
- bool [enableStop](#)

*Enables or disables WDOG in stop mode.*
- bool [enableDebug](#)

*Enables or disables WDOG in debug mode.*

### 41.3.2 struct wdog\_config\_t

#### Data Fields

- bool [enableWdog](#)

*Enables or disables WDOG.*
- [wdog\\_clock\\_source\\_t](#) [clockSource](#)

*Clock source select.*
- [wdog\\_clock\\_prescaler\\_t](#) [prescaler](#)

*Clock prescaler value.*
- [wdog\\_work\\_mode\\_t](#) [workMode](#)

*Configures WDOG work mode in debug stop and wait mode.*
- bool [enableUpdate](#)

## Enumeration Type Documentation

- `bool enableInterrupt`  
*Enables or disables WDOG interrupt.*
- `bool enableWindowMode`  
*Enables or disables WDOG window mode.*
- `uint32_t windowValue`  
*Window value.*
- `uint32_t timeoutValue`  
*Timeout value.*

### 41.3.3 `struct wdog_test_config_t`

#### Data Fields

- `wdog_test_mode_t testMode`  
*Selects test mode.*
- `wdog_tested_byte_t testedByte`  
*Selects tested byte in byte test mode.*
- `uint32_t timeoutValue`  
*Timeout value.*

## 41.4 Macro Definition Documentation

### 41.4.1 `#define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

## 41.5 Enumeration Type Documentation

### 41.5.1 `enum wdog_clock_source_t`

Enumerator

*kWDOG\_LpoClockSource* WDOG clock sourced from LPO.

*kWDOG\_AlternateClockSource* WDOG clock sourced from alternate clock source.

### 41.5.2 `enum wdog_clock_prescaler_t`

Enumerator

*kWDOG\_ClockPrescalerDivide1* Divided by 1.  
*kWDOG\_ClockPrescalerDivide2* Divided by 2.  
*kWDOG\_ClockPrescalerDivide3* Divided by 3.  
*kWDOG\_ClockPrescalerDivide4* Divided by 4.  
*kWDOG\_ClockPrescalerDivide5* Divided by 5.  
*kWDOG\_ClockPrescalerDivide6* Divided by 6.  
*kWDOG\_ClockPrescalerDivide7* Divided by 7.

*kWDOG\_ClockPrescalerDivide8* Divided by 8.

#### 41.5.3 enum wdog\_test\_mode\_t

Enumerator

*kWDOG\_QuickTest* Selects quick test.

*kWDOG\_Bytetest* Selects byte test.

#### 41.5.4 enum wdog\_tested\_byte\_t

Enumerator

*kWDOG\_TestByte0* Byte 0 selected in byte test mode.

*kWDOG\_TestByte1* Byte 1 selected in byte test mode.

*kWDOG\_TestByte2* Byte 2 selected in byte test mode.

*kWDOG\_TestByte3* Byte 3 selected in byte test mode.

#### 41.5.5 enum \_wdog\_interrupt\_enable\_t

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

Enumerator

*kWDOG\_InterruptEnable* WDOG timeout generates an interrupt before reset.

#### 41.5.6 enum \_wdog\_status\_flags\_t

This structure contains the WDOG status flags for use in the WDOG functions.

Enumerator

*kWDOG\_RunningFlag* Running flag, set when WDOG is enabled.

*kWDOG\_TimeoutFlag* Interrupt flag, set when an exception occurs.

### 41.6 Function Documentation

#### 41.6.1 void WDOG\_GetDefaultConfig ( wdog\_config\_t \* config )

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

## Function Documentation

```
* wdogConfig->enableWdog = true;
* wdogConfig->clockSource = kWDOG_IpoClockSource;
* wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
* wdogConfig->workMode.enableWait = true;
* wdogConfig->workMode.enableStop = false;
* wdogConfig->workMode.enableDebug = false;
* wdogConfig->enableUpdate = true;
* wdogConfig->enableInterrupt = false;
* wdogConfig->enableWindowMode = false;
* wdogConfig->windowValue = 0;
* wdogConfig->timeoutValue = 0xFFFFU;
*
```

### Parameters

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

### See Also

[wdog\\_config\\_t](#)

### 41.6.2 void WDOG\_Init ( **WDOG\_Type** \* *base*, **const wdog\_config\_t** \* *config* )

This function initializes the WDOG. When called, the WDOG runs according to the configuration. To reconfigure WDOG without forcing a reset first, enableUpdate must be set to true in the configuration.

This is an example.

```
* wdog_config_t config;
* WDOG_GetDefaultConfig(&config);
* config.timeoutValue = 0x7ffU;
* config.enableUpdate = true;
* WDOG_Init(wdog_base,&config);
*
```

### Parameters

|               |                              |
|---------------|------------------------------|
| <i>base</i>   | WDOG peripheral base address |
| <i>config</i> | The configuration of WDOG    |

### 41.6.3 void WDOG\_Deinit ( **WDOG\_Type** \* *base* )

This function shuts down the WDOG. Ensure that the WDOG\_STCTRLH.ALLOWUPDATE is 1 which indicates that the register update is enabled.

#### 41.6.4 void WDOG\_SetTestModeConfig ( WDOG\_Type \* *base*, wdog\_test\_config\_t \* *config* )

This function is used to configure the WDOG functional test. When called, the WDOG goes into test mode and runs according to the configuration. Ensure that the WDOG\_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

This is an example.

```
* wdog_test_config_t test_config;
* test_config.testMode = kWDOG_QuickTest;
* test_config.timeoutValue = 0xffffffffu;
* WDOG_SetTestModeConfig(wdog_base, &test_config);
*
```

Parameters

|               |                                           |
|---------------|-------------------------------------------|
| <i>base</i>   | WDOG peripheral base address              |
| <i>config</i> | The functional test configuration of WDOG |

#### 41.6.5 static void WDOG\_Enable ( WDOG\_Type \* *base* ) [inline], [static]

This function write value into WDOG\_STCTRLH register to enable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

#### 41.6.6 static void WDOG\_Disable ( WDOG\_Type \* *base* ) [inline], [static]

This function writes a value into the WDOG\_STCTRLH register to disable the WDOG. It is a write-once register. Ensure that the WCT window is still open and that register has not been written to in this WCT while the function is called.

Parameters

## Function Documentation

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

### 41.6.7 static void WDOG\_EnableInterrupts ( **WDOG\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

This function writes a value into the WDOG\_STCTRLH register to enable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

|             |                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | WDOG 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"><li>• kWDOG_InterruptEnable</li></ul> |

### 41.6.8 static void WDOG\_DisableInterrupts ( **WDOG\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

This function writes a value into the WDOG\_STCTRLH register to disable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

|             |                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | WDOG 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"><li>• kWDOG_InterruptEnable</li></ul> |

### 41.6.9 **uint32\_t** WDOG\_GetStatusFlags ( **WDOG\_Type** \* *base* )

This function gets all status flags.

This is an example for getting the Running Flag.

```
* uint32_t status;
* status = WDOG_GetStatusFlags (wdog_base) &
* kWDOG_RunningFlag;
```

\*

## Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

## Returns

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

## See Also

[\\_wdog\\_status\\_flags\\_t](#)

- true: a related status flag has been set.
- false: a related status flag is not set.

**41.6.10 void WDOG\_ClearStatusFlags ( WDOG\_Type \* *base*, uint32\_t *mask* )**

This function clears the WDOG status flag.

This is an example for clearing the timeout (interrupt) flag.

```
* WDOG_ClearStatusFlags (wdog_base, kWDOG_TimeoutFlag);
*
```

## Parameters

|             |                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------|
| <i>base</i> | WDOG peripheral base address                                                                                 |
| <i>mask</i> | The status flags to clear. The parameter could be any combination of the following values. kWDOG_TimeoutFlag |

**41.6.11 static void WDOG\_SetTimeoutValue ( WDOG\_Type \* *base*, uint32\_t *timeoutCount* ) [inline], [static]**

This function sets the timeout value. It should be ensured that the time-out value for the WDOG is always greater than 2xWCT time + 20 bus clock cycles. This function writes a value into WDOG\_TOVALH and WDOG\_TOVALL registers which are write-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

## Function Documentation

Parameters

|                     |                                               |
|---------------------|-----------------------------------------------|
| <i>base</i>         | WDOG peripheral base address                  |
| <i>timeoutCount</i> | WDOG timeout value; count of WDOG clock tick. |

### 41.6.12 static void WDOG\_SetWindowValue ( WDOG\_Type \* *base*, uint32\_t *windowValue* ) [inline], [static]

This function sets the WDOG window value. This function writes a value into WDOG\_WINH and WDOG\_WINL registers which are write-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

|                    |                              |
|--------------------|------------------------------|
| <i>base</i>        | WDOG peripheral base address |
| <i>windowValue</i> | WDOG window value.           |

### 41.6.13 static void WDOG\_Unlock ( WDOG\_Type \* *base* ) [inline], [static]

This function unlocks the WDOG register written. Before starting the unlock sequence and following configuration, disable the global interrupts. Otherwise, an interrupt may invalidate the unlocking sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

### 41.6.14 void WDOG\_Refresh ( WDOG\_Type \* *base* )

This function feeds the WDOG. This function should be called before the WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

41.6.15 **static uint16\_t WDOG\_GetResetCount( WDOG\_Type \* *base* ) [inline], [static]**

This function gets the WDOG reset count value.

## Function Documentation

Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

Returns

WDOG reset count value.

### 41.6.16 static void WDOG\_ClearResetCount( WDOG\_Type \* *base* ) [inline], [static]

This function clears the WDOG reset count value.

Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | WDOG peripheral base address |
|-------------|------------------------------|

# Chapter 42

## Clock Driver

### 42.1 Overview

The KSDK provides APIs for Kinetis devices clock operation.

### 42.2 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.

### 42.3 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 **FSL\_SDK\_DISABLE\_DRIVER\_CLOCK\_CONTROL** 0  
*Configure whether driver controls clock.*
- #define **MCG\_INTERNAL\_IRC\_48M** 48000000U  
*IRC48M clock frequency in Hz.*
- #define **DMAMUX\_CLOCKS**  
*Clock ip name array for DMAMUX.*
- #define **RTC\_CLOCKS**  
*Clock ip name array for RTC.*
- #define **ENET\_CLOCKS**  
*Clock ip name array for ENET.*
- #define **PORT\_CLOCKS**  
*Clock ip name array for PORT.*
- #define **SAI\_CLOCKS**  
*Clock ip name array for SAI.*
- #define **FLEXBUS\_CLOCKS**  
*Clock ip name array for FLEXBUS.*
- #define **TSI\_CLOCKS**  
*Clock ip name array for TSI.*
- #define **LPUART\_CLOCKS**  
*Clock ip name array for LPUART.*
- #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 **SDHC\_CLOCKS**  
*Clock ip name array for SDHC.*
- #define **FTM\_CLOCKS**  
*Clock ip name array for FTM.*
- #define **EDMA\_CLOCKS**  
*Clock ip name array for EDMA.*
- #define **FLEXCAN\_CLOCKS**  
*Clock ip name array for FLEXCAN.*
- #define **DAC\_CLOCKS**  
*Clock ip name array for DAC.*
- #define **ADC16\_CLOCKS**  
*Clock ip name array for ADC16.*
- #define **SDRAM\_CLOCKS**  
*Clock ip name array for SDRAM.*
- #define **MPU\_CLOCKS**  
*Clock ip name array for MPU.*
- #define **VREF\_CLOCKS**

- *Clock ip name array for VREF.*  
• #define **CMT\_CLOCKS**
  - *Clock ip name array for CMT.*  
• #define **UART\_CLOCKS**
  - *Clock ip name array for UART.*  
• #define **TPM\_CLOCKS**
  - *Clock ip name array for TPM.*  
• #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 **FTF\_CLOCKS**
  - *Clock ip name array for FTF.*  
• #define **CMP\_CLOCKS**
  - *Clock ip name array for CMP.*  
• #define **LPO\_CLK\_FREQ** 1000U
  - *LPO clock frequency.*  
• #define **SYS\_CLK** kCLOCK\_CoreSysClk
- Peripherals clock source definition.*

## Enumerations

- enum **clock\_name\_t** {
   
kCLOCK\_CoreSysClk,
   
kCLOCK\_PlatClk,
   
kCLOCK\_BusClk,
   
kCLOCK\_FlexBusClk,
   
kCLOCK\_FlashClk,
   
kCLOCK\_FastPeriphClk,
   
kCLOCK\_PllFllSelClk,
   
kCLOCK\_Er32kClk,
   
kCLOCK\_Osc0ErClk,
   
kCLOCK\_Osc1ErClk,
   
kCLOCK\_Osc0ErClkUndiv,
   
kCLOCK\_McgFixedFreqClk,
   
kCLOCK\_McgInternalRefClk,
   
kCLOCK\_McgFllClk,
   
kCLOCK\_McgPll0Clk,
   
kCLOCK\_McgPll1Clk,
   
kCLOCK\_McgExtPllClk,
   
kCLOCK\_McgPeriphClk,
   
kCLOCK\_McgIrc48MClk,
   
kCLOCK\_LpoClk
 }

*Clock name used to get clock frequency.*

## External clock frequency

- enum `clock_usb_src_t`{  
    kCLOCK\_UsbSrcPll0 = SIM\_SOPT2\_USBSRC(1U) | SIM\_SOPT2\_PLLFLLSEL(1U),  
    kCLOCK\_UsbSrcUsbPfd = SIM\_SOPT2\_USBSRC(1U) | SIM\_SOPT2\_PLLFLLSEL(2U),  
    kCLOCK\_UsbSrcIrc48M = SIM\_SOPT2\_USBSRC(1U) | SIM\_SOPT2\_PLLFLLSEL(3U),  
    kCLOCK\_UsbSrcExt = SIM\_SOPT2\_USBSRC(0U),  
    kCLOCK\_UsbSrcUnused = 0xFFFFFFFFU }
- USB clock source definition.
- enum `clock_usb_phy_src_t` { kCLOCK\_UsbPhySrcExt = 0U }
- Source of the USB HS PHY.
- enum `clock_usb_pfd_src_t`{  
    kCLOCK\_UsbPfdSrcExt = 0U,  
    kCLOCK\_UsbPfdSrcFracDivBy4 = 1U,  
    kCLOCK\_UsbPfdSrcFracDivBy2 = 2U,  
    kCLOCK\_UsbPfdSrcFrac = 3U }
- Source of the USB HS PFD clock (USB1PFDCLK)
- 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\_EREFSTEN\_MASK }
- OSCERCLK enable mode.
- enum `mcg_fll_src_t`{  
    kMCG\_FllSrcExternal,  
    kMCG\_FllSrcInternal }
- MCG FLL reference clock source select.
- enum `mcg_irc_mode_t`{  
    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\_PlRefOsc0,  
 kMCG\_PlRefOsc1 }
- 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,  
 kMCG\_OscselIrc }
- MCG OSC Clock Select.*
- enum `mcg_pll_clk_select_t` { kMCG\_PlClkSelPlI0 }
- 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\_Pl0LostFlag = (1U << 5U),  
 kMCG\_Pl0LockFlag = (1U << 6U),  
 kMCG\_ExtPl0LostFlag = (1U << 9U) }
- MCG status flags.*

## External clock frequency

- enum `_mcg_irclk_enable_mode` {  
    `kMCG_IrclkEnable` = MCG\_C1\_IRCLKEN\_MASK,  
    `kMCG_IrclkEnableInStop` = MCG\_C1\_IREFSTEN\_MASK }  
    *MCG internal reference clock (MCGIRCLK) enable mode definition.*
- enum `_mcg_pll_enable_mode` {  
    `kMCG_PllEnableIndependent` = MCG\_C5\_PLLCLKEN0\_MASK,  
    `kMCG_PllEnableInStop` = MCG\_C5\_PLLSTEN0\_MASK }  
    *MCG PLL clock enable mode definition.*
- enum `mcg_mode_t` {  
    `kMCG_ModeFEI` = 0U,  
    `kMCG_ModeFBI`,  
    `kMCG_ModeBLPI`,  
    `kMCG_ModeFEE`,  
    `kMCG_ModeFBE`,  
    `kMCG_ModeBLPE`,  
    `kMCG_ModePBE`,  
    `kMCG_ModePEE`,  
    `kMCG_ModeError` }  
    *MCG mode definitions.*

## Functions

- static void `CLOCK_EnableClock` (`clock_ip_name_t` name)  
    *Enable the clock for specific IP.*
- static void `CLOCK_DisableClock` (`clock_ip_name_t` name)  
    *Disable the clock for specific IP.*
- static void `CLOCK_SetEr32kClock` (`uint32_t` src)  
    *Set ERCLK32K source.*
- static void `CLOCK_SetSdhc0Clock` (`uint32_t` src)  
    *Set SDHC0 clock source.*
- static void `CLOCK_SetEnetTime0Clock` (`uint32_t` src)  
    *Set enet timestamp clock source.*
- static void `CLOCK_SetRmii0Clock` (`uint32_t` src)  
    *Set RMII clock source.*
- static void `CLOCK_SetLpuartClock` (`uint32_t` src)  
    *Set LPUART clock source.*
- static void `CLOCK_SetTpmClock` (`uint32_t` src)  
    *Set TPM clock source.*
- static void `CLOCK_SetTraceClock` (`uint32_t` src, `uint32_t` divValue, `uint32_t` fracValue)  
    *Set debug trace clock source.*
- static void `CLOCK_SetPlifllSelClock` (`uint32_t` src, `uint32_t` divValue, `uint32_t` fracValue)  
    *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_EnableUsbhs0Clock` (`clock_usb_src_t` src, `uint32_t` freq)  
    *Enable USB HS clock.*
- void `CLOCK_DisableUsbhs0Clock` (`void`)

- *Disable USB HS clock.*
- `bool CLOCK_EnableUsbhs0PhyPllClock (clock_usb_phy_src_t src, uint32_t freq)`  
*Enable USB HS PHY PLL clock.*
- `void CLOCK_DisableUsbhs0PhyPllClock (void)`  
*Disable USB HS PHY PLL clock.*
- `void CLOCK_EnableUsbhs0PfdClock (uint8_t frac, clock_usb_pfd_src_t src)`  
*Enable USB HS PFD clock.*
- `void CLOCK_DisableUsbhs0PfdClock (void)`  
*Disable USB HS PFD clock.*
- `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 outdiv3, uint32_t outdiv4)`  
*System clock divider.*
- `uint32_t CLOCK_GetFreq (clock_name_t clockName)`  
*Gets the clock frequency for a specific clock name.*
- `uint32_t CLOCK_GetCoreSysClkFreq (void)`  
*Get the core clock or system clock frequency.*
- `uint32_t CLOCK_GetPlatClkFreq (void)`  
*Get the platform clock frequency.*
- `uint32_t CLOCK_GetBusClkFreq (void)`  
*Get the bus clock frequency.*
- `uint32_t CLOCK_GetFlexBusClkFreq (void)`  
*Get the flexbus 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[PLLFLLSEL].*
- `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).*
- `uint32_t CLOCK_GetOsc0ErClkUndivFreq (void)`  
*Get the OSC0 external reference undivided clock frequency (OSC0ERCLK\_UNDIV).*
- `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))`

## External clock frequency

*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.*
- `uint32_t CLOCK_GetExtPllFreq (void)`  
*Gets the MCG external PLL frequency.*
- `void CLOCK_SetExtPllFreq (uint32_t freq)`  
*Sets the MCG external PLL 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 frdiv)`  
*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.*
- `void CLOCK_SetPllClkSel (mcg_pll_clk_select_t pllcs)`  
*Set the PLL selection.*

### 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.*
- `void CLOCK_SetExtPllMonitorMode (mcg_monitor_mode_t mode)`  
*Sets the external PLL 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, oscer\_config\_t const \*config)  
*Configures the OSC external reference clock (OSCERCLK).*
- static void **OSC\_SetCapLoad** (OSC\_Type \*base, uint8\_t capLoad)  
*Sets the capacitor load configuration for the oscillator.*
- void **CLOCK\_InitOsc0** (osc\_config\_t const \*config)  
*Initializes the OSC0.*
- void **CLOCK\_DeinitOsc0** (void)  
*Deinitializes the OSC0.*

## External clock frequency

- static void **CLOCK\_SetXtal0Freq** (uint32\_t freq)  
*Sets the XTAL0 frequency based on board settings.*
- static void **CLOCK\_SetXtal32Freq** (uint32\_t freq)  
*Sets the XTAL32/RTC\_CLKIN frequency based on board settings.*

## MCG auto-trim machine.

- status\_t **CLOCK\_TrimInternalRefClk** (uint32\_t extFreq, uint32\_t desireFreq, uint32\_t \*actualFreq, mcg\_atm\_select\_t atms)  
*Auto trims the internal reference clock.*

## MCG mode functions.

- mcg\_mode\_t **CLOCK\_GetMode** (void)  
*Gets the current MCG mode.*
- status\_t **CLOCK\_SetFeiMode** (mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FEI mode.*
- status\_t **CLOCK\_SetFeeMode** (uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FEE mode.*
- status\_t **CLOCK\_SetFbiMode** (mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FBI mode.*
- status\_t **CLOCK\_SetFbeMode** (uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FBE mode.*
- status\_t **CLOCK\_SetBlpiMode** (void)  
*Sets the MCG to BLPI mode.*
- status\_t **CLOCK\_SetBlpeMode** (void)  
*Sets the MCG to BLPE mode.*
- status\_t **CLOCK\_SetPbeMode** (mcg\_pll\_clk\_select\_t pllcs, mcg\_pll\_config\_t const \*config)  
*Sets the MCG to PBE mode.*
- status\_t **CLOCK\_SetPeeMode** (void)  
*Sets the MCG to PEE mode.*
- status\_t **CLOCK\_ExternalModeToFbeModeQuick** (void)

## Data Structure Documentation

- status\_t **CLOCK\_InternalModeToFbiModeQuick** (void)  
*Switches the MCG to FBE mode from the external mode.*
- status\_t **CLOCK\_BootToFeiMode** (mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Switches the MCG to FBI mode from internal modes.*
- status\_t **CLOCK\_BootToFeeMode** (mcg\_oscsel\_t oscsel, uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FEI mode during system boot up.*
- status\_t **CLOCK\_BootToBlpiMode** (uint8\_t fcrdiv, mcg\_irc\_mode\_t ircs, uint8\_t ircEnableMode)  
*Sets the MCG to BLPI mode during system boot up.*
- status\_t **CLOCK\_BootToBlpeMode** (mcg\_oscsel\_t oscsel)  
*Sets the MCG to BLPE mode during system boot up.*
- status\_t **CLOCK\_BootToPeeMode** (mcg\_oscsel\_t oscsel, mcg\_pll\_clk\_select\_t pllcs, mcg\_pll\_config\_t const \*config)  
*Sets the MCG to PEE mode during system boot up.*
- status\_t **CLOCK\_SetMcgConfig** (mcg\_config\_t const \*config)  
*Sets the MCG to a target mode.*

## 42.4 Data Structure Documentation

### 42.4.1 struct sim\_clock\_config\_t

#### Data Fields

- uint8\_t **pllFllSel**  
*PLL/FLL/IRC48M selection.*
- uint8\_t **pllFllDiv**  
*PLLSEL clock divider divisor.*
- uint8\_t **pllFllFrac**  
*PLLSEL clock divider fraction.*
- uint8\_t **er32kSrc**  
*ERCLK32K source selection.*
- uint32\_t **clkdiv1**  
*SIM\_CLKDIV1.*

**42.4.1.0.0.97 Field Documentation**

- 42.4.1.0.0.97.1 `uint8_t sim_clock_config_t::plIFIISel`
- 42.4.1.0.0.97.2 `uint8_t sim_clock_config_t::plIFIIDiv`
- 42.4.1.0.0.97.3 `uint8_t sim_clock_config_t::plIFIIFrac`
- 42.4.1.0.0.97.4 `uint8_t sim_clock_config_t::er32kSrc`
- 42.4.1.0.0.97.5 `uint32_t sim_clock_config_t::clkdiv1`

**42.4.2 struct oscer\_config\_t****Data Fields**

- `uint8_t enableMode`  
*OSCERCLK enable mode.*
- `uint8_t erclkDiv`  
*Divider for OSCERCLK.*

**42.4.2.0.0.98 Field Documentation**

- 42.4.2.0.0.98.1 `uint8_t oscer_config_t::enableMode`

OR'ed value of `_oscer_enable_mode`.

- 42.4.2.0.0.98.2 `uint8_t oscer_config_t::erclkDiv`

**42.4.3 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

### 42.4.3.0.0.99 Field Documentation

42.4.3.0.0.99.1 `uint32_t osc_config_t::freq`

42.4.3.0.0.99.2 `uint8_t osc_config_t::capLoad`

42.4.3.0.0.99.3 `osc_mode_t osc_config_t::workMode`

42.4.3.0.0.99.4 `oscer_config_t osc_config_t::oscerConfig`

### 42.4.4 `struct mcg_pll_config_t`

#### Data Fields

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

### 42.4.4.0.0.100 Field Documentation

42.4.4.0.0.100.1 `uint8_t mcg_pll_config_t::enableMode`

OR'ed value of `_mcg_pll_enable_mode`.

42.4.4.0.0.100.2 `uint8_t mcg_pll_config_t::prdiv`

42.4.4.0.0.100.3 `uint8_t mcg_pll_config_t::vdiv`

### 42.4.5 `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.*
- `mcg_pll_clk_select_t pllcS`  
*PLL select as output, PLLCS.*

#### 42.4.5.0.0.101 Field Documentation

**42.4.5.0.0.101.1 mcg\_mode\_t mcg\_config\_t::mcgMode**

**42.4.5.0.0.101.2 uint8\_t mcg\_config\_t::irclkEnableMode**

**42.4.5.0.0.101.3 mcg\_irc\_mode\_t mcg\_config\_t::ircs**

**42.4.5.0.0.101.4 uint8\_t mcg\_config\_t::fcrdiv**

**42.4.5.0.0.101.5 uint8\_t mcg\_config\_t::frdiv**

**42.4.5.0.0.101.6 mcg\_drs\_t mcg\_config\_t::drs**

**42.4.5.0.0.101.7 mcg\_dmx32\_t mcg\_config\_t::dmx32**

**42.4.5.0.0.101.8 mcg\_oscsel\_t mcg\_config\_t::oscsel**

**42.4.5.0.0.101.9 mcg\_pll\_config\_t mcg\_config\_t::pll0Config**

**42.4.5.0.0.101.10 mcg\_pll\_clk\_select\_t mcg\_config\_t::pllcS**

### 42.5 Macro Definition Documentation

#### 42.5.1 #define FSL\_SDK\_DISABLE\_DRIVER\_CLOCK\_CONTROL 0

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

Note

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

## Macro Definition Documentation

**42.5.2 #define FSL\_CLOCK\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 1))**

**42.5.3 #define MCG\_INTERNAL\_IRC\_48M 48000000U**

**42.5.4 #define DMAMUX\_CLOCKS**

**Value:**

```
{ \
 kCLOCK_Dmamux0 \
}
```

**42.5.5 #define RTC\_CLOCKS**

**Value:**

```
{ \
 kCLOCK_Rtc0 \
}
```

**42.5.6 #define ENET\_CLOCKS**

**Value:**

```
{ \
 kCLOCK_Enet0 \
}
```

**42.5.7 #define PORT\_CLOCKS**

**Value:**

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

**42.5.8 #define SAI\_CLOCKS**

**Value:**

```
{ \
 kCLOCK_Sai0 \
}
```

#### 42.5.9 #define FLEXBUS\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Flexbus0 \
}
```

#### 42.5.10 #define TSI\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Tsi0 \
}
```

#### 42.5.11 #define LPUART\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Lpuart0 \
}
```

#### 42.5.12 #define EWM\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Ewm0 \
}
```

#### 42.5.13 #define PIT\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Pit0 \
}
```

## Macro Definition Documentation

### 42.5.14 #define DSPI\_CLOCKS

**Value:**

```
{
 kCLOCK_Spi0, kCLOCK_Spi1, kCLOCK_Spi2 \
}
```

### 42.5.15 #define LPTMR\_CLOCKS

**Value:**

```
{
 kCLOCK_Lptmr0 \
}
```

### 42.5.16 #define SDHC\_CLOCKS

**Value:**

```
{
 kCLOCK_Sdhc0 \
}
```

### 42.5.17 #define FTM\_CLOCKS

**Value:**

```
{
 kCLOCK_Ftm0, kCLOCK_Ftm1, kCLOCK_Ftm2, kCLOCK_Ftm3 \
}
```

### 42.5.18 #define EDMA\_CLOCKS

**Value:**

```
{
 kCLOCK_Dma0 \
}
```

#### 42.5.19 #define FLEXCAN\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Flexcan0, kCLOCK_Flexcan1 \
}
```

#### 42.5.20 #define DAC\_CLOCKS

**Value:**

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

#### 42.5.21 #define ADC16\_CLOCKS

**Value:**

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

#### 42.5.22 #define SDRAM\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Sdramc0 \
}
```

#### 42.5.23 #define MPU\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Mpu0 \
}
```

## Macro Definition Documentation

### 42.5.24 #define VREF\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Vref0 \
}
```

### 42.5.25 #define CMT\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Cmt0 \
}
```

### 42.5.26 #define UART\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Uart0, kCLOCK_Uart1, kCLOCK_Uart2, kCLOCK_Uart3, kCLOCK_Uart4 \
}
```

### 42.5.27 #define TPM\_CLOCKS

**Value:**

```
{ \
 kCLOCK_TpmInvalid, kCLOCK_Tpm1, kCLOCK_Tpm2 \
}
```

### 42.5.28 #define RNGA\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Rnga0 \
}
```

#### 42.5.29 #define CRC\_CLOCKS

**Value:**

```
{
 kCLOCK_Crc0 \
}
```

#### 42.5.30 #define I2C\_CLOCKS

**Value:**

```
{
 kCLOCK_I2c0, kCLOCK_I2c1, kCLOCK_I2c2, kCLOCK_I2c3 \
}
```

#### 42.5.31 #define PDB\_CLOCKS

**Value:**

```
{
 kCLOCK_Pdb0 \
}
```

#### 42.5.32 #define FTF\_CLOCKS

**Value:**

```
{
 kCLOCK_Ftf0 \
}
```

#### 42.5.33 #define CMP\_CLOCKS

**Value:**

```
{
 kCLOCK_Cmp0, kCLOCK_Cmp1, kCLOCK_Cmp2, kCLOCK_Cmp3 \
}
```

## Enumeration Type Documentation

### 42.5.34 #define SYS\_CLK kCLOCK\_CoreSysClk

## 42.6 Enumeration Type Documentation

### 42.6.1 enum clock\_name\_t

Enumerator

*kCLOCK\_CoreSysClk* Core/system clock.  
*kCLOCK\_PlatClk* Platform clock.  
*kCLOCK\_BusClk* Bus clock.  
*kCLOCK\_FlexBusClk* FlexBus clock.  
*kCLOCK\_FlashClk* Flash clock.  
*kCLOCK\_FastPeriphClk* Fast peripheral clock.  
*kCLOCK\_PllFllSelClk* The clock after SIM[PLLPLLSEL].  
*kCLOCK\_Er32kClk* External reference 32K clock (ERCLK32K)  
*kCLOCK\_Osc0ErClk* OSC0 external reference clock (OSC0ERCLK)  
*kCLOCK\_Osc1ErClk* OSC1 external reference clock (OSC1ERCLK)  
*kCLOCK\_Osc0ErClkUndiv* OSC0 external reference undivided clock(OSC0ERCLK\_UNDIV).  
*kCLOCK\_McgFixedFreqClk* MCG fixed frequency clock (MCGFFCLK)  
*kCLOCK\_McgInternalRefClk* MCG internal reference clock (MCGIRCLK)  
*kCLOCK\_McgFllClk* MCGFLLCLK.  
*kCLOCK\_McgPll0Clk* MCGPLL0CLK.  
*kCLOCK\_McgPll1Clk* MCGPLL1CLK.  
*kCLOCK\_McgExtPllClk* EXT\_PLLCLK.  
*kCLOCK\_McgPeriphClk* MCG peripheral clock (MCGPCLK)  
*kCLOCK\_McgIrc48MClk* MCG IRC48M clock.  
*kCLOCK\_LpoClk* LPO clock.

### 42.6.2 enum clock\_usb\_src\_t

Enumerator

*kCLOCK\_UsbSrcPll0* Use PLL0.  
*kCLOCK\_UsbSrcUsbPfd* Use USBPFDCLK.  
*kCLOCK\_UsbSrcIrc48M* Use IRC48M.  
*kCLOCK\_UsbSrcExt* Use USB\_CLKIN.  
*kCLOCK\_UsbSrcUnused* Used when the function does not care the clock source.

### 42.6.3 enum clock\_usb\_phy\_src\_t

Enumerator

*kCLOCK\_UsbPhySrcExt* Use external crystal.

**42.6.4 enum clock\_usb\_pfd\_src\_t**

Enumerator

- kCLOCK\_UsbPfdSrcExt* Use external crystal.
- kCLOCK\_UsbPfdSrcFracDivBy4* Use PFD\_FRAC output divided by 4.
- kCLOCK\_UsbPfdSrcFracDivBy2* Use PFD\_FRAC output divided by 2.
- kCLOCK\_UsbPfdSrcFrac* Use PFD\_FRAC output.

**42.6.5 enum clock\_ip\_name\_t****42.6.6 enum osc\_mode\_t**

Enumerator

- kOSC\_ModeExt* Use an external clock.
- kOSC\_ModeOscLowPower* Oscillator low power.
- kOSC\_ModeOscHighGain* Oscillator high gain.

**42.6.7 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

**42.6.8 enum \_oscer\_enable\_mode**

Enumerator

- kOSC\_ErClkEnable* Enable.
- kOSC\_ErClkEnableInStop* Enable in stop mode.

**42.6.9 enum mcg\_fll\_src\_t**

Enumerator

- kMCG\_FllSrcExternal* External reference clock is selected.
- kMCG\_FllSrcInternal* The slow internal reference clock is selected.

## Enumeration Type Documentation

### 42.6.10 enum mcg\_irc\_mode\_t

Enumerator

*kMCG\_IrcSlow* Slow internal reference clock selected.

*kMCG\_IrcFast* Fast internal reference clock selected.

### 42.6.11 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.

### 42.6.12 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.

### 42.6.13 enum mcg\_pll\_ref\_src\_t

Enumerator

*kMCG\_PliRefOsc0* Selects OSC0 as PLL reference clock.

*kMCG\_PliRefOsc1* Selects OSC1 as PLL reference clock.

### 42.6.14 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.

**42.6.15 enum mcg\_atm\_select\_t**

Enumerator

*kMCG\_AtmSel32k* 32 kHz Internal Reference Clock selected*kMCG\_AtmSel4m* 4 MHz Internal Reference Clock selected**42.6.16 enum mcg\_oscsel\_t**

Enumerator

*kMCG\_OscselOsc* Selects System Oscillator (OSCCLK)*kMCG\_OscselRtc* Selects 32 kHz RTC Oscillator.*kMCG\_OscselIrc* Selects 48 MHz IRC Oscillator.**42.6.17 enum mcg\_pll\_clk\_select\_t**

Enumerator

*kMCG\_PlClkSelPll0* PLL0 output clock is selected.**42.6.18 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.**42.6.19 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.

## Enumeration Type Documentation

### 42.6.20 enum \_mcg\_status\_flags\_t

Enumerator

*kMCG\_Osc0LostFlag* OSC0 lost.  
*kMCG\_Osc0InitFlag* OSC0 crystal initialized.  
*kMCG\_RtcOscLostFlag* RTC OSC lost.  
*kMCG\_Pl0LostFlag* PLL0 lost.  
*kMCG\_Pl0LockFlag* PLL0 locked.  
*kMCG\_ExtPl0LostFlag* External PLL lost.

### 42.6.21 enum \_mcg\_irclk\_enable\_mode

Enumerator

*kMCG\_IrclkEnable* MCGIRCLK enable.  
*kMCG\_IrclkEnableInStop* MCGIRCLK enable in stop mode.

### 42.6.22 enum \_mcg\_pll\_enable\_mode

Enumerator

*kMCG\_PlEnableIndependent* 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\_PlEnableInStop* MCGPLLCLK enable in STOP mode.

### 42.6.23 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.

## 42.7 Function Documentation

42.7.1 **static void CLOCK\_EnableClock( clock\_ip\_name\_t *name* ) [inline], [static]**

## Function Documentation

Parameters

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

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

Parameters

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

### 42.7.3 static void CLOCK\_SetEr32kClock ( *uint32\_t src* ) [inline], [static]

Parameters

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

### 42.7.4 static void CLOCK\_SetSdhc0Clock ( *uint32\_t src* ) [inline], [static]

Parameters

|            |                                      |
|------------|--------------------------------------|
| <i>src</i> | The value to set SDHC0 clock source. |
|------------|--------------------------------------|

### 42.7.5 static void CLOCK\_SetEnetTime0Clock ( *uint32\_t src* ) [inline], [static]

Parameters

|            |                                               |
|------------|-----------------------------------------------|
| <i>src</i> | The value to set enet timestamp clock source. |
|------------|-----------------------------------------------|

### 42.7.6 static void CLOCK\_SetRmii0Clock ( *uint32\_t src* ) [inline], [static]

Parameters

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

#### 42.7.7 static void CLOCK\_SetLpuartClock ( *uint32\_t src* ) [inline], [static]

Parameters

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

#### 42.7.8 static void CLOCK\_SetTpmClock ( *uint32\_t src* ) [inline], [static]

Parameters

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

#### 42.7.9 static void CLOCK\_SetTraceClock ( *uint32\_t src, uint32\_t divValue, uint32\_t fracValue* ) [inline], [static]

Parameters

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

#### 42.7.10 static void CLOCK\_SetPLLSELClock ( *uint32\_t src, uint32\_t divValue, uint32\_t fracValue* ) [inline], [static]

Parameters

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

#### 42.7.11 static void CLOCK\_SetClkOutClock ( *uint32\_t src* ) [inline], [static]

## Function Documentation

Parameters

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

### 42.7.12 static void CLOCK\_SetRtcClkOutClock ( uint32\_t *src* ) [inline], [static]

Parameters

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

### 42.7.13 bool CLOCK\_EnableUsbhs0Clock ( clock\_usb\_src\_t *src*, uint32\_t *freq* )

This function only enables the access to USB HS prepheral, upper layer should first call the [CLOCK\\_EnableUsbhs0PhyPllClock](#) to enable the PHY clock to use USB HS.

Parameters

|             |                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------|
| <i>src</i>  | USB HS does not care about the clock source, here must be <a href="#">kCLOCK_UsbSrcUnused</a> . |
| <i>freq</i> | USB HS does not care about the clock source, so this parameter is ignored.                      |

Return values

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

### 42.7.14 void CLOCK\_DisableUsbhs0Clock ( void )

Disable USB HS clock, this function should not be called after [CLOCK\\_DisableUsbhs0PhyPllClock](#).

### 42.7.15 bool CLOCK\_EnableUsbhs0PhyPllClock ( clock\_usb\_phy\_src\_t *src*, uint32\_t *freq* )

This function enables the internal 480MHz USB PHY PLL clock.

Parameters

|             |                                         |
|-------------|-----------------------------------------|
| <i>src</i>  | USB HS PHY PLL 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 HS clock. |

#### 42.7.16 void CLOCK\_DisableUsbhs0PhyPllClock ( void )

This function disables USB HS PHY PLL clock.

#### 42.7.17 void CLOCK\_EnableUsbhs0PfdClock ( uint8\_t *frac*, clock\_usb\_pfd\_src\_t *src* )

This function enables USB HS PFD clock. It should be called after function [CLOCK\\_EnableUsbhs0PhyPllClock](#). The PFD output clock is selected by the parameter *src*. When the *src* is [kCLOCK\\_UsbPfdSrcExt](#), then the PFD output is from external crystal directly, in this case, the *frac* is not used. In other cases, the PFD\_FRAC output clock frequency is 480MHz\*18/*frac*, the PFD output frequency is based on the PFD\_FRAC output.

Parameters

|             |                                                                 |
|-------------|-----------------------------------------------------------------|
| <i>frac</i> | The value set to PFD_FRAC, it must be in the range of 18 to 35. |
| <i>src</i>  | Source of the USB HS PFD clock (USB1PFDCLK).                    |

#### 42.7.18 void CLOCK\_DisableUsbhs0PfdClock ( void )

This function disables USB HS PFD clock. It should be called before function [CLOCK\\_DisableUsbhs0PhyPllClock](#).

#### 42.7.19 bool CLOCK\_EnableUsbfs0Clock ( clock\_usb\_src\_t *src*, uint32\_t *freq* )

## Function Documentation

Parameters

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

Return values

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

### 42.7.20 static void CLOCK\_DisableUsbfs0Clock ( void ) [inline], [static]

Disable USB FS clock.

### 42.7.21 static void CLOCK\_SetOutDiv ( uint32\_t *outdiv1*, uint32\_t *outdiv2*, uint32\_t *outdiv3*, uint32\_t *outdiv4* ) [inline], [static]

Set the SIM\_CLKDIV1[OUTDIV1], SIM\_CLKDIV1[OUTDIV2], SIM\_CLKDIV1[OUTDIV3], SIM\_CLKDIV1[OUTDIV4].

Parameters

|                |                               |
|----------------|-------------------------------|
| <i>outdiv1</i> | Clock 1 output divider value. |
| <i>outdiv2</i> | Clock 2 output divider value. |
| <i>outdiv3</i> | Clock 3 output divider value. |
| <i>outdiv4</i> | Clock 4 output divider value. |

### 42.7.22 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 <code>clock_name_t</code> |
|------------------|--------------------------------------------------|

Returns

Clock frequency value in Hertz

#### **42.7.23 `uint32_t CLOCK_GetCoreSysClkFreq( void )`**

Returns

Clock frequency in Hz.

#### **42.7.24 `uint32_t CLOCK_GetPlatClkFreq( void )`**

Returns

Clock frequency in Hz.

#### **42.7.25 `uint32_t CLOCK_GetBusClkFreq( void )`**

Returns

Clock frequency in Hz.

#### **42.7.26 `uint32_t CLOCK_GetFlexBusClkFreq( void )`**

Returns

Clock frequency in Hz.

#### **42.7.27 `uint32_t CLOCK_GetFlashClkFreq( void )`**

Returns

Clock frequency in Hz.

## Function Documentation

### 42.7.28 `uint32_t CLOCK_GetPllFllSelClkFreq( void )`

Returns

Clock frequency in Hz.

### 42.7.29 `uint32_t CLOCK_GetEr32kClkFreq( void )`

Returns

Clock frequency in Hz.

### 42.7.30 `uint32_t CLOCK_GetOsc0ErClkFreq( void )`

Returns

Clock frequency in Hz.

### 42.7.31 `uint32_t CLOCK_GetOsc0ErClkUndivFreq( void )`

Returns

Clock frequency in Hz.

### 42.7.32 `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. |
|---------------|-------------------------------------|

### 42.7.33 `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. |
|---------------|-------------------------------------|

#### 42.7.34 `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.

#### 42.7.35 `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.

#### 42.7.36 `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.

#### 42.7.37 `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 MCFFCLK.

## Function Documentation

### 42.7.38 `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.

### 42.7.39 `uint32_t CLOCK_GetExtPllFreq( void )`

This function gets the MCG external PLL frequency in Hz.

Returns

The frequency of the MCG external PLL.

### 42.7.40 `void CLOCK_SetExtPllFreq( uint32_t freq )`

This function sets the MCG external PLL frequency in Hz. The MCG external PLL frequency is passed to the MCG driver using this function. Call this function after the external PLL frequency is changed. Otherwise, the APIs, which are used to get the frequency, may return an incorrect value.

Parameters

|            |                                |
|------------|--------------------------------|
| <i>The</i> | frequency of MCG external PLL. |
|------------|--------------------------------|

### 42.7.41 `static void CLOCK_SetLowPowerEnable( bool enable ) [inline], [static]`

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

Parameters

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

#### 42.7.42 status\_t CLOCK\_SetInternalRefClkConfig ( uint8\_t *enableMode*, mcg\_ircl\_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 <a href="#">_mcg_irclk_enable_mode</a> . |
| <i>ircs</i>       | MCGIRCLK clock source, choose fast or slow.                                   |
| <i>fcrdiv</i>     | Fast IRC divider setting (FCRDIV).                                            |

Return values

|                               |                                                                                                                                      |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>kStatus_MCG_SourceUsed</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.                                                                                        |

#### 42.7.43 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/BLPE/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_SourceUsed</i> | Because the external reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs. |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|

## Function Documentation

|                        |                                            |
|------------------------|--------------------------------------------|
| <i>kStatus_Success</i> | External reference clock set successfully. |
|------------------------|--------------------------------------------|

### 42.7.44 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]. |
|--------------|----------------------------------------------------------------|

### 42.7.45 void CLOCK\_EnablePll0 ( mcg\_pll\_config\_t const \* *config* )

This function sets us 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. |
|---------------|-----------------------------------------|

### 42.7.46 static void CLOCK\_DisablePll0 ( void ) [inline], [static]

This function disables the PLL0 in FLL mode. It should be used together with the [CLOCK\\_EnablePll0](#).

### 42.7.47 uint32\_t CLOCK\_CalcPllDiv ( uint32\_t *refFreq*, uint32\_t *desireFreq*, uint8\_t \* *prdiv*, uint8\_t \* *vdiv* )

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

Parameters

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

Returns

Closest frequency match that the PLL was able generate.

#### 42.7.48 void CLOCK\_SetPIIClkSel ( mcg\_pll\_clk\_select\_t *pllcs* )

This function sets the PLL selection between PLL0/PLL1/EXTPLL, and waits for change finished.

Parameters

|              |                    |
|--------------|--------------------|
| <i>pllcs</i> | The PLL to select. |
|--------------|--------------------|

#### 42.7.49 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. |
|-------------|----------------------|

#### 42.7.50 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. |
|-------------|----------------------|

#### 42.7.51 void CLOCK\_SetPll0MonitorMode ( mcg\_monitor\_mode\_t *mode* )

This function sets the PLL0 clock monitor mode. See [mcg\\_monitor\\_mode\\_t](#) for details.

## Function Documentation

Parameters

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

### 42.7.52 void CLOCK\_SetExtPllMonitorMode ( mcg\_monitor\_mode\_t *mode* )

This function sets the external PLL clock monitor mode. See [mcg\\_monitor\\_mode\\_t](#) for details.

Parameters

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

### 42.7.53 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](#).

### 42.7.54 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 <a href="#">_mcg_status_flags_t</a> . |
|-------------|---------------------------------------------------------------------------------------------------------------------|

#### 42.7.55 static void OSC\_SetExtRefClkConfig ( **OSC\_Type** \* *base*, **oscer\_config\_t** const \* *config* ) [inline], [static]

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

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

OSC_SetExtRefClkConfig(OSC, &config);
```

Parameters

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

#### 42.7.56 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 <a href="#">_osc_cap_load</a> . |

Example:

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

#### 42.7.57 void CLOCK\_InitOsc0 ( **osc\_config\_t** const \* *config* )

This function initializes the OSC0 according to the board configuration.

## Function Documentation

Parameters

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

### 42.7.58 void CLOCK\_DeinitOsc0( void )

This function deinitializes the OSC0.

### 42.7.59 static void CLOCK\_SetXtal0Freq( uint32\_t *freq* ) [inline], [static]

Parameters

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

### 42.7.60 static void CLOCK\_SetXtal32Freq( uint32\_t *freq* ) [inline], [static]

Parameters

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

### 42.7.61 status\_t CLOCK\_TrimInternalRefClk( uint32\_t *extFreq*, uint32\_t *desireFreq*, uint32\_t \* *actualFreq*, mcg\_atm\_select\_t *atms* )

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

Parameters

|                   |                                                        |
|-------------------|--------------------------------------------------------|
| <i>extFreq</i>    | External clock frequency, which should be a bus clock. |
| <i>desireFreq</i> | Frequency to trim to.                                  |
| <i>actualFreq</i> | Actual frequency after trimming.                       |

|             |                                             |
|-------------|---------------------------------------------|
| <i>atms</i> | Trim fast or slow internal reference clock. |
|-------------|---------------------------------------------|

Return values

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

#### 42.7.62 **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](#).

#### 42.7.63 **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

## Function Documentation

|                                     |                                           |
|-------------------------------------|-------------------------------------------|
| <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.

### 42.7.64 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>fllStableDelay</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. |

### 42.7.65 status\_t CLOCK\_SetFbiMode ( mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*)(void) fllStableDelay )

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>fllStableDelay</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. |

Note

If *dmx32* is set to *kMCG\_Dmx32Fine*, the slow IRC must not be trimmed to frequency above 32768 Hz.

#### 42.7.66 **status\_t CLOCK\_SetFbeMode ( uint8\_t *frdiv*, mcg\_dmx32\_t *dmx32*, mcg\_drs\_t *drs*, void(\*)(void) *fllStableDelay* )**

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. |
|-------------------------------------|--------------------------------------|

## Function Documentation

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

### 42.7.67 **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. |

### 42.7.68 **status\_t CLOCK\_SetBlpeMode ( void )**

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

Return values

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

### 42.7.69 **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 `pllc`s 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_Pl-ClkSelExtPll, NULL);`

#### 42.7.70 `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.

#### 42.7.71 `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(...);
*
```

## Function Documentation

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. |

### 42.7.72 status\_t CLOCK\_InternalModeToFbiModeQuick ( void )

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

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

Return values

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

### 42.7.73 status\_t CLOCK\_BootToFeiMode ( mcg\_dmx32\_t *dmx32*, mcg\_drs\_t *drs*, void(\*)(void) *fllStableDelay* )

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

Parameters

|                       |                                                  |
|-----------------------|--------------------------------------------------|
| <i>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. |

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.

#### 42.7.74 **status\_t CLOCK\_BootToFeeMode ( mcg\_oscsel\_t oscsel, uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*)(void) fllStableDelay )**

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

## Parameters

|                       |                                                  |
|-----------------------|--------------------------------------------------|
| <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. |

#### 42.7.75 **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 system boot up.

## Function Documentation

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 <a href="#"><u>_mcg_irclk_enable_mode</u></a> . |

Return values

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

### 42.7.76 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 system boot up.

Parameters

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

Return values

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

### 42.7.77 status\_t CLOCK\_BootToPeeMode ( mcg\_oscsel\_t *oscsel*, mcg\_pll\_clk\_select\_t *pllcs*, mcg\_pll\_config\_t const \* *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. |

#### 42.7.78 **status\_t CLOCK\_SetMcgConfig ( mcg\_config\_t const \* 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.

## 42.8 Variable Documentation

### 42.8.1 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(80000000); // 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.

## Variable Documentation

### 42.8.2 `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.

## 42.9 Multipurpose Clock Generator (MCG)

The KSDK provides a peripheral driver for the MCG module of Kinetis devices.

### 42.9.1 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.

#### 42.9.1.1 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\\_GetPll0Freq\(\)](#), [CLOCK\\_GetPll1Freq\(\)](#), and [CLOCK\\_GetExtPllFreq\(\)](#). These functions get the clock frequency based on the current MCG registers.

#### 42.9.1.2 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 driver. 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 micro seconds 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\\_EnablePll0\(\)](#) and [CLOCK\\_EnablePll1\(\)](#). To enable the MCGPLLCLK, the PLL reference clock divider(PRDIV) and the PLL VCO divider(VDIV) must be set to a proper value. The function [CLOCK\\_CalcPllDiv\(\)](#) helps to get the PRDIV/VDIV.

## Multipurpose Clock Generator (MCG)

### 42.9.1.3 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.

### 42.9.1.4 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.

### 42.9.1.5 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.

### 42.9.1.6 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.

## 42.9.2 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.

### 42.9.2.1 Switch between BLPI and FEI

| Use case    | Steps                      | Functions                                                                |
|-------------|----------------------------|--------------------------------------------------------------------------|
| BLPI -> FEI | BLPI -> FBI                | <code>CLOCK_InternalModeToFbi-ModeQuick(...)</code>                      |
|             | FBI -> FEI                 | <code>CLOCK_SetFeiMode(...)</code>                                       |
|             | Configure MCGIRCLK if need | <code>CLOCK_SetInternalRefClk-Config(...)</code>                         |
| FEI -> BLPI | Configure MCGIRCLK if need | <code>CLOCK_SetInternalRefClk-Config(...)</code>                         |
|             | FEI -> FBI                 | <code>CLOCK_SetFbiMode(...)</code> with <code>fllStableDelay=NULL</code> |
|             | FBI -> BLPI                | <code>CLOCK_SetLowPower-Enable(true)</code>                              |

### 42.9.2.2 Switch between BLPI and FEE

| Use case    | Steps                                | Functions                                                                |
|-------------|--------------------------------------|--------------------------------------------------------------------------|
| BLPI -> FEE | BLPI -> FBI                          | <code>CLOCK_InternalModeToFbi-ModeQuick(...)</code>                      |
|             | Change external clock source if need | <code>CLOCK_SetExternalRefClk-Config(...)</code>                         |
|             | FBI -> FEE                           | <code>CLOCK_SetFeeMode(...)</code>                                       |
| FEE -> BLPI | Configure MCGIRCLK if need           | <code>CLOCK_SetInternalRefClk-Config(...)</code>                         |
|             | FEE -> FBI                           | <code>CLOCK_SetFbiMode(...)</code> with <code>fllStableDelay=NULL</code> |
|             | FBI -> BLPI                          | <code>CLOCK_SetLowPower-Enable(true)</code>                              |

## Multipurpose Clock Generator (MCG)

### 42.9.2.3 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(...) // f1l-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 f1lStableDelay=NULL |
|             | FBI -> BLPI                          | CLOCK_SetLowPower-Enable(true)                 |

### 42.9.2.4 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 -> PEE | BLPE -> FBE | CLOCK_ExternalModeToFbe-ModeQuick(...) |

|             |               |                                               |
|-------------|---------------|-----------------------------------------------|
|             | 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)                 |

#### 42.9.2.5 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 flStableDelay=NULL |
|             | Change source | CLOCK_SetExternalRefClkConfig(...)            |
|             | FBI -> FEE    | CLOCK_SetFeeMode(...)                         |
| FEE -> BLPE | FEE -> FBI    | CLOCK_SetFbiMode(...) with flStableDelay=NULL |
|             | Change source | CLOCK_SetExternalRefClkConfig(...)            |
|             | PBI -> FBE    | CLOCK_SetFbeMode(...) with flStableDelay=NULL |
|             | FBE -> BLPE   | CLOCK_SetLowPowerEnable(true)                 |

### 42.9.2.6 Switch between BLPI and PEI

| Use case    | Steps                      | Functions                             |
|-------------|----------------------------|---------------------------------------|
| BLPI -> PEI | BLPI -> PBI                | CLOCK_SetPbiMode(...)                 |
|             | PBI -> PEI                 | CLOCK_SetPeiMode(...)                 |
|             | Configure MCGIRCLK if need | CLOCK_SetInternalRefClkConfig(...)    |
| PEI -> BLPI | Configure MCGIRCLK if need | CLOCK_SetInternalRefClkConfig         |
|             | PEI -> FBI                 | CLOCK_InternalModeToFbiModeQuick(...) |
|             | FBI -> BLPI                | CLOCK_SetLowPowerEnable(true)         |

# Chapter 43

## Debug Console

### 43.1 Overview

This part describes the programming interface of the debug console driver. The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data.

### 43.2 Function groups

#### 43.2.1 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);
```

### 43.2.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype " %[flags][width][.precision][length]specifier", which is explained below

| flags   | Description                                                                                                                                                                                                                                                                                                                                                                             |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -       | Left-justified within the given field width. Right-justified is the default.                                                                                                                                                                                                                                                                                                            |
| +       | Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.                                                                                                                                                                                                                                |
| (space) | If no sign is written, a blank space is inserted before the value.                                                                                                                                                                                                                                                                                                                      |
| #       | Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed. |
| 0       | Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).                                                                                                                                                                                                                                                                           |

| Width    | Description                                                                                                                                                                                            |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (number) | A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger. |
| *        | The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.                                                          |

| .precision | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .number    | For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed. |
| .*         | The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

| length         | Description |
|----------------|-------------|
| Do not support |             |

| specifier | Description                                  |
|-----------|----------------------------------------------|
| d or i    | Signed decimal integer                       |
| f         | Decimal floating point                       |
| F         | Decimal floating point capital letters       |
| x         | Unsigned hexadecimal integer                 |
| X         | Unsigned hexadecimal integer capital letters |
| o         | Signed octal                                 |
| b         | Binary value                                 |
| p         | Pointer address                              |
| u         | Unsigned decimal integer                     |
| c         | Character                                    |
| s         | String of characters                         |
| n         | Nothing printed                              |

## Function groups

- Support a format specifier for SCANF following this prototype " %[\*][width][length]specifier", which is explained below

| *                                                                                                                                                                | Description |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| An optional starting asterisk indicates that the data is to be read from the stream but ignored. In other words, it is not stored in the corresponding argument. |             |

| width                                                                                        | Description |
|----------------------------------------------------------------------------------------------|-------------|
| This specifies the maximum number of characters to be read in the current reading operation. |             |

| length      | Description                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| hh          | The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).                                                                     |
| h           | The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).                                                                    |
| l           | The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.           |
| ll          | The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s. |
| L           | The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).                                                                                            |
| j or z or t | Not supported                                                                                                                                                                                           |

| specifier | Qualifying Input                                                                                                                                                                                                                                 | Type of argument |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| c         | Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end. | char *           |

| specifier              | Qualifying Input                                                                                                                                                                                                            | Type of argument |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| i                      | Integer: : Number optionally preceded with a + or - sign                                                                                                                                                                    | int *            |
| d                      | Decimal integer: Number optionally preceded with a + or - sign                                                                                                                                                              | int *            |
| a, A, e, E, f, F, g, G | Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4 | float *          |
| o                      | Octal Integer:                                                                                                                                                                                                              | int *            |
| s                      | String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).                                                                  | char *           |
| u                      | Unsigned decimal integer.                                                                                                                                                                                                   | unsigned int *   |

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file.

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the KSDK printf/scanf.

```
#if SDK_DEBUGCONSOLE /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF DbgConsole_Printf
#define SCANF DbgConsole_Scanf
#define PUTCHAR DbgConsole_Putchar
#define GETCHAR DbgConsole_Getchar
#else /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF printf
#define SCANF scanf
#define PUTCHAR putchar
#define GETCHAR getchar
#endif /* SDK_DEBUGCONSOLE */
```

### 43.3 Typical use case

#### Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

## Typical use case

### Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalents 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\r\nTime: %u ticks %2.5f milliseconds\r\nDONE\r", "1 day", 86400, 86.4);
```

### Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

### Print out failure messages using KSDK \_\_assert\_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
 PRINTF("ASSERT ERROR \"% %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file,
 line, func);
 for (;;)
 {}
}
```

### Note:

To use 'printf' and 'scanf' for GNUC Base, add file '**fsl\_sbrk.c**' in path: ..\{package}\devices\{subset}\utilities\fsl-sbrk.c to your project.

## Modules

- Semihosting

## 43.4 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.

### 43.4.1 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.

### 43.4.2 Guide Semihosting for Keil µVision

**NOTE:** Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

#### Step 1: Prepare code

Remove function `fputc` and `fgetc` is used to support KEIL in "fsl\_debug\_console.c" and add the following code to project.

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY; /* used for Debug Input */
```

```
struct __FILE
{
 int handle;
};

FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
 return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{ /* blocking */
 while (ITM_CheckChar() != 1)
 ;
 return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
 /* Your implementation of ferror */
 return EOF;
}

void _ttywrch(int ch)
{
 ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
 goto label; /* endless loop */
}
```

### Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click.
2. Select "Target" tab and not select "Use MicroLIB".
3. Select "Debug" tab, select "J-Link/J-Trace Cortex" and click "Setting button".
4. Select "Debug" tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK.

### Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

### Step 4: Building the project

1. Choose "Debug" on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

### 43.4.3 Guide Semihosting for KDS

**NOTE:** After the setting use "printf" for debugging.

#### Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select “Libraries” on “Cross ARM C Linker” and delete “nosys”.
3. Select “Miscellaneous” on “Cross ARM C Linker”, add “-specs=rdimon.specs” to “Other link flags” and tick “Use newlib-nano”, and click OK.

#### Step 2: Building the project

1. In menu bar, choose Project>Build Project.

#### Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick “Enable semihosting and Telnet”. Press “Apply” and “Debug”.
2. After clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

### 43.4.4 Guide Semihosting for ATL

**NOTE:** J-Link has to be used to enable semihosting.

#### Step 1: Prepare code

Add the following code to the project.

```
int _write(int file, char *ptr, int len)
{
 /* Implement your write code here. This is used by puts and printf. */
 int i=0;
 for(i=0 ; i<len ; i++)
 ITM_SendChar((*ptr++));
 return len;
}
```

#### Step 2: Setting up the environment

1. In menu bar, choose Debug Configurations. In tab "Embedded C/C++ Application" choose "- Semihosting\_ATL\_xxx debug J-Link".
2. In tab "Debugger" set up as follows.
  - JTAG mode must be selected

## Semihosting

- SWV tracing must be enabled
  - Enter the Core Clock frequency, which is hardware board-specific.
  - Enter the desired SWO Clock frequency. The latter depends on the JTAG Probe and must be a multiple of the Core Clock value.
3. Click "Apply" and "Debug".

### Step 3: Starting semihosting

1. In the Views menu, expand the submenu SWV and open the docking view "SWV Console". 2. Open the SWV settings panel by clicking the "Configure Serial Wire Viewer" button in the SWV Console view toolbar. 3. Configure the data ports to be traced by enabling the ITM channel 0 check-box in the ITM stimulus ports group: Choose "EXETRC: Trace Exceptions" and In tab "ITM Stimulus Ports" choose "Enable Port" 0. Then click "OK".
2. It is recommended not to enable other SWV trace functionalities at the same time because this may over use the SWO pin causing packet loss due to a limited bandwidth (certain other SWV tracing capabilities can send a lot of data at very high-speed). Save the SWV configuration by clicking the OK button. The configuration is saved with other debug configurations and remains effective until changed.
3. Press the red Start/Stop Trace button to send the SWV configuration to the target board to enable SWV trace recording. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be present, if the configuration registers on the target board are reset. Also, tracing does not start until the target starts to execute.
4. Start the target execution again by pressing the green Resume Debug button.
5. The SWV console now shows the printf() output.

### 43.4.5 Guide Semihosting for ARMGCC

#### Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
  - "Host Name (or IP address)" : localhost
  - "Port" :2333
  - "Connection type" : Telnet.
  - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

#### Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
```

```
defsym=__heap_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__heap_size__=0x2000")
```

## Step 2: Building the project

1. Change "CMakeLists.txt":

**Change** "SET(CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE "\${CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE} -specs=nano.specs")"  
**to** "SET(CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE "\${CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE} -specs=rdimon.specs")"

### Replace paragraph

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mapcs")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
--gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
muldefs")
```

### To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
```

## Semihosting

```
G} --specs=rdimon.specs ")
Remove
target_link_libraries(semihosting_ARMGCC.elf debug nosys)
2. Run "build_debug.bat" to build project
```

### Step 3: Starting semihosting

- (a) Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

- (b) After the setting, press "enter". The PuTTY window now shows the printf() output.

# Chapter 44

## Notification Framework

### 44.1 Overview

This section describes the programming interface of the Notifier driver.

### 44.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

These are the steps for the configuration transition.

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.  
The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending a "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.
2. After the "BEFORE" message is processed successfully, the system switches to the new configuration.
3. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This example shows how to use the Notifier in the Power Manager application.

```
#include "fsl_notifier.h"

/* Definition of the Power Manager callback */
status_t callback0(notifier_notification_block_t *notify, void *data)
{
 status_t ret = kStatus_Success;

 ...
 ...

 return ret;
}
/* Definition of the Power Manager user function */
status_t APP_PowerModeSwitch(notifier_user_config_t *targetConfig, void *userData)
```

## Notifier Overview

# Data Structures

- struct **notifier\_notification\_block\_t**  
*notification block passed to the registered callback function. [More...](#)*
  - struct **notifier\_callback\_config\_t**  
*Callback configuration structure. [More...](#)*
  - struct **notifier\_handle\_t**  
*Notifier handle structure. [More...](#)*

## TypeDefs

- `typedef void notifier_user_config_t`  
*Notifier user configuration type.*
  - `typedef status_t(* notifier_user_function_t )(notifier_user_config_t *targetConfig, void *userData)`  
*Notifier user function prototype Use this function to execute specific operations in configuration switch.*

- `typedef status_t(* notifier_callback_t )(notifier_notification_block_t *notify, void *data)`  
*Callback prototype.*

## Enumerations

- `enum _notifier_status {`  
 `kStatus_NOTIFIER_ErrorNotificationBefore,`  
 `kStatus_NOTIFIER_ErrorNotificationAfter }`  
*Notifier error codes.*
- `enum notifier_policy_t {`  
 `kNOTIFIER_PolicyAgreement,`  
 `kNOTIFIER_PolicyForcible }`  
*Notifier policies.*
- `enum notifier_notification_type_t {`  
 `kNOTIFIER_NotifyRecover = 0x00U,`  
 `kNOTIFIER_NotifyBefore = 0x01U,`  
 `kNOTIFIER_NotifyAfter = 0x02U }`  
*Notification type.*
- `enum notifier_callback_type_t {`  
 `kNOTIFIER_CallbackBefore = 0x01U,`  
 `kNOTIFIER_CallbackAfter = 0x02U,`  
 `kNOTIFIER_CallbackBeforeAfter = 0x03U }`  
*The callback type, which indicates kinds of notification the callback handles.*

## Functions

- `status_t NOTIFIER_CreateHandle (notifier_handle_t *notifierHandle, notifier_user_config_t **configs, uint8_t configsNumber, notifier_callback_config_t *callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void *userData)`  
*Creates a Notifier handle.*
- `status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)`  
*Switches the configuration according to a pre-defined structure.*
- `uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)`  
*This function returns the last failed notification callback.*

## 44.3 Data Structure Documentation

### 44.3.1 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

### 44.3.1.0.0.102 Field Documentation

44.3.1.0.0.102.1 `notifier_user_config_t* notifier_notification_block_t::targetConfig`

44.3.1.0.0.102.2 `notifier_policy_t notifier_notification_block_t::policy`

44.3.1.0.0.102.3 `notifier_notification_type_t notifier_notification_block_t::notifyType`

### 44.3.2 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.*

### 44.3.2.0.0.103 Field Documentation

44.3.2.0.0.103.1 `notifier_callback_t notifier_callback_config_t::callback`

44.3.2.0.0.103.2 `notifier_callback_type_t notifier_callback_config_t::callbackType`

44.3.2.0.0.103.3 `void* notifier_callback_config_t::callbackData`

### 44.3.3 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.*

#### 44.3.3.0.0.104 Field Documentation

44.3.3.0.0.104.1 `notifier_user_config_t** notifier_handle_t::configsTable`

44.3.3.0.0.104.2 `uint8_t notifier_handle_t::configsNumber`

44.3.3.0.0.104.3 `notifier_callback_config_t* notifier_handle_t::callbacksTable`

44.3.3.0.0.104.4 `uint8_t notifier_handle_t::callbacksNumber`

44.3.3.0.0.104.5 `uint8_t notifier_handle_t::errorCallbackIndex`

44.3.3.0.0.104.6 `uint8_t notifier_handle_t::currentConfigIndex`

44.3.3.0.0.104.7 `notifier_user_function_t notifier_handle_t::userFunction`

44.3.3.0.0.104.8 `void* notifier_handle_t::userData`

### 44.4 Typedef Documentation

#### 44.4.1 `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.

#### 44.4.2 `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.

### 44.4.3 **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.

## 44.5 Enumeration Type Documentation

### 44.5.1 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.

#### 44.5.2 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.

#### 44.5.3 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.

#### 44.5.4 enum notifier\_callback\_type\_t

Used in the callback configuration structure (`notifier_callback_config_t`) to specify when the registered callback is called during configuration switch initiated by the `NOTIFIER_SwitchConfig()`. Callback can be invoked in following situations.

- Before the configuration switch (Callback return value can affect `NOTIFIER_SwitchConfig()` execution. See the `NOTIFIER_SwitchConfig()` and `notifier_policy_t` documentation).
- After an unsuccessful attempt to switch configuration
- After a successful configuration switch

Enumerator

***kNOTIFIER\_CallbackBefore*** Callback handles BEFORE notification.

***kNOTIFIER\_CallbackAfter*** Callback handles AFTER notification.

***kNOTIFIER\_CallbackBeforeAfter*** Callback handles BEFORE and AFTER notification.

## Function Documentation

### 44.6 Function Documentation

44.6.1 `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.

#### 44.6.2 **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.

### 44.6.3 **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 45

## Shell

### 45.1 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.

### 45.2 Function groups

#### 45.2.1 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>> ");
```

#### 45.2.2 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.                                  |

### 45.2.3 Shell Operation

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
SHELL_Main(&user_context);
```

## Data Structures

- struct [p\\_shell\\_context\\_t](#)  
*Data structure for Shell environment.* [More...](#)
- struct [shell\\_command\\_context\\_t](#)  
*User command data structure.* [More...](#)
- struct [shell\\_command\\_context\\_list\\_t](#)  
*Structure list command.* [More...](#)

## Macros

- #define [SHELL\\_USE\\_HISTORY](#) (0U)  
*Macro to set on/off history feature.*
- #define [SHELL\\_SEARCH\\_IN\\_HIST](#) (1U)  
*Macro to set on/off history feature.*
- #define [SHELL\\_USE\\_FILE\\_STREAM](#) (0U)  
*Macro to select method stream.*
- #define [SHELL\\_AUTO\\_COMPLETE](#) (1U)  
*Macro to set on/off auto-complete feature.*
- #define [SHELL\\_BUFFER\\_SIZE](#) (64U)  
*Macro to set console buffer size.*
- #define [SHELL\\_MAX\\_ARGS](#) (8U)  
*Macro to set maximum arguments in command.*
- #define [SHELL\\_HIST\\_MAX](#) (3U)  
*Macro to set maximum count of history commands.*
- #define [SHELL\\_MAX\\_CMD](#) (6U)  
*Macro to set maximum count of commands.*

## Typedefs

- typedef void(\* [send\\_data\\_cb\\_t](#) )(uint8\_t \*buf, uint32\_t len)  
*Shell user send data callback prototype.*
- typedef void(\* [recv\\_data\\_cb\\_t](#) )(uint8\_t \*buf, uint32\_t len)  
*Shell user receiver data callback prototype.*
- typedef int(\* [printf\\_data\\_t](#) )(const char \*format,...)

- *Shell user printf data prototype.*  
**typedef int32\_t(\* cmd\_function\_t )(p\_shell\_context\_t context, int32\_t argc, char \*\*argv)**  
*User command function prototype.*

## Enumerations

- **enum fun\_key\_status\_t {**  
**kSHELL\_Normal = 0U,**  
**kSHELL\_Special = 1U,**  
**kSHELL\_Function = 2U }**  
*A type for the handle special key.*

## Shell functional operation

- **void SHELL\_Init (p\_shell\_context\_t context, send\_data\_cb\_t send\_cb, recv\_data\_cb\_t recv\_cb, printf\_data\_t shell\_printf, char \*prompt)**  
*Enables the clock gate and configures the Shell module according to the configuration structure.*
- **int32\_t SHELL\_RegisterCommand (const shell\_command\_context\_t \*command\_context)**  
*Shell register command.*
- **int32\_t SHELL\_Main (p\_shell\_context\_t context)**  
*Main loop for Shell.*

## 45.3 Data Structure Documentation

### 45.3.1 struct shell\_context\_struct

#### Data Fields

- **char \* prompt**  
*Prompt string.*
- **enum \_fun\_key\_status stat**  
*Special key status.*
- **char line [SHELL\_BUFFER\_SIZE]**  
*Consult buffer.*
- **uint8\_t cmd\_num**  
*Number of user commands.*
- **uint8\_t l\_pos**  
*Total line position.*
- **uint8\_t c\_pos**  
*Current line position.*
- **send\_data\_cb\_t send\_data\_func**  
*Send data interface operation.*
- **recv\_data\_cb\_t recv\_data\_func**  
*Receive data interface operation.*
- **uint16\_t hist\_current**  
*Current history command in hist buff.*
- **uint16\_t hist\_count**  
*Total history command in hist buff.*
- **char hist\_buf [SHELL\_HIST\_MAX][SHELL\_BUFFER\_SIZE]**

## Data Structure Documentation

- *History buffer.*  
• bool [exit](#)  
*Exit Flag.*

### 45.3.2 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.*

#### 45.3.2.0.0.105 Field Documentation

##### 45.3.2.0.0.105.1 const char\* shell\_command\_context\_t::pcCommand

For example "help". It must be all lower case.

##### 45.3.2.0.0.105.2 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".

##### 45.3.2.0.0.105.3 const cmd\_function\_t shell\_command\_context\_t::pFuncCallBack

##### 45.3.2.0.0.105.4 uint8\_t shell\_command\_context\_t::cExpectedNumberOfParameters

### 45.3.3 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.*

## 45.4 Macro Definition Documentation

45.4.1 `#define SHELL_USE_HISTORY (0U)`

45.4.2 `#define SHELL_SEARCH_IN_HIST (1U)`

45.4.3 `#define SHELL_USE_FILE_STREAM (0U)`

45.4.4 `#define SHELL_AUTO_COMPLETE (1U)`

45.4.5 `#define SHELL_BUFFER_SIZE (64U)`

45.4.6 `#define SHELL_MAX_ARGS (8U)`

45.4.7 `#define SHELL_HIST_MAX (3U)`

45.4.8 `#define SHELL_MAX_CMD (6U)`

## 45.5 Typedef Documentation

45.5.1 `typedef void(* send_data_cb_t)(uint8_t *buf, uint32_t len)`

45.5.2 `typedef void(* recv_data_cb_t)(uint8_t *buf, uint32_t len)`

45.5.3 `typedef int(* printf_data_t)(const char *format,...)`

45.5.4 `typedef int32_t(* cmd_function_t)(p_shell_context_t context, int32_t argc, char **argv)`

## 45.6 Enumeration Type Documentation

45.6.1 `enum fun_key_status_t`

Enumerator

*kSHELL\_Normal* Normal key.

*kSHELL\_Special* Special key.

*kSHELL\_Function* Function key.

## Function Documentation

### 45.7 Function Documentation

#### 45.7.1 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                               |

#### 45.7.2 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

#### 45.7.3 int32\_t SHELL\_Main ( *p\_shell\_context\_t context* )

Main loop for Shell; After this function is called, Shell begins to initialize the basic variables and starts to work.

### Parameters

|                |                                                          |
|----------------|----------------------------------------------------------|
| <i>context</i> | The pointer to the Shell environment and runtime states. |
|----------------|----------------------------------------------------------|

### Returns

This function does not return until Shell command exit was called.

## Function Documentation

**How to Reach Us:**

**Home Page:**

[nxp.com](http://nxp.com)

**Web Support:**

[nxp.com/support](http://nxp.com/support)

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

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

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

© 2016 Freescale Semiconductor, Inc.

