

MCUXpresso SDK API Reference Manual

NXP Semiconductors

Document Number: MCUXSDKK64FAPIRM
Rev. 0
Dec 2018



Contents

Chapter Introduction

Chapter Driver errors status

Chapter Architectural Overview

Chapter Trademarks

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

5.1	Overview	11
5.2	Typical use case	11
5.2.1	Polling Configuration	11
5.2.2	Interrupt Configuration	11
5.3	Data Structure Documentation	14
5.3.1	struct adc16_config_t	14
5.3.2	struct adc16_hardware_compare_config_t	15
5.3.3	struct adc16_channel_config_t	15
5.4	Macro Definition Documentation	16
5.4.1	FSL_ADC16_DRIVER_VERSION	16
5.5	Enumeration Type Documentation	16
5.5.1	_adc16_channel_status_flags	16
5.5.2	_adc16_status_flags	16
5.5.3	adc16_channel_mux_mode_t	16
5.5.4	adc16_clock_divider_t	17
5.5.5	adc16_resolution_t	17
5.5.6	adc16_clock_source_t	17
5.5.7	adc16_long_sample_mode_t	17
5.5.8	adc16_reference_voltage_source_t	18
5.5.9	adc16_hardware_average_mode_t	18
5.5.10	adc16_hardware_compare_mode_t	18
5.6	Function Documentation	18
5.6.1	ADC16_Init	18

Section Number	Title	Page Number
5.6.2	ADC16_Deinit	19
5.6.3	ADC16_GetDefaultConfig	19
5.6.4	ADC16_DoAutoCalibration	19
5.6.5	ADC16_SetOffsetValue	20
5.6.6	ADC16_EnableDMA	20
5.6.7	ADC16_EnableHardwareTrigger	20
5.6.8	ADC16_SetChannelMuxMode	21
5.6.9	ADC16_SetHardwareCompareConfig	21
5.6.10	ADC16_SetHardwareAverage	21
5.6.11	ADC16_GetStatusFlags	22
5.6.12	ADC16_ClearStatusFlags	22
5.6.13	ADC16_SetChannelConfig	22
5.6.14	ADC16_GetChannelConversionValue	24
5.6.15	ADC16_GetChannelStatusFlags	24

Chapter CMP: Analog Comparator Driver

6.1	Overview	25
6.2	Typical use case	25
6.2.1	Polling Configuration	25
6.2.2	Interrupt Configuration	25
6.3	Data Structure Documentation	27
6.3.1	struct cmp_config_t	27
6.3.2	struct cmp_filter_config_t	27
6.3.3	struct cmp_dac_config_t	28
6.4	Macro Definition Documentation	28
6.4.1	FSL_CMP_DRIVER_VERSION	28
6.5	Enumeration Type Documentation	28
6.5.1	_cmp_interrupt_enable	28
6.5.2	_cmp_status_flags	28
6.5.3	cmp_hysteresis_mode_t	29
6.5.4	cmp_reference_voltage_source_t	29
6.6	Function Documentation	29
6.6.1	CMP_Init	29
6.6.2	CMP_Deinit	29
6.6.3	CMP_Enable	30
6.6.4	CMP_GetDefaultConfig	30
6.6.5	CMP_SetInputChannels	30
6.6.6	CMP_EnableDMA	31
6.6.7	CMP_EnableWindowMode	31

Section Number	Title	Page Number
6.6.8	CMP_EnablePassThroughMode	31
6.6.9	CMP_SetFilterConfig	32
6.6.10	CMP_SetDACCConfig	32
6.6.11	CMP_EnableInterrupts	32
6.6.12	CMP_DisableInterrupts	32
6.6.13	CMP_GetStatusFlags	33
6.6.14	CMP_ClearStatusFlags	33
 Chapter CMT: Carrier Modulator Transmitter Driver		
7.1	Overview	35
7.2	Clock formulas	35
7.3	Typical use case	35
7.4	Data Structure Documentation	37
7.4.1	struct cmt_modulate_config_t	37
7.4.2	struct cmt_config_t	38
7.5	Macro Definition Documentation	39
7.5.1	FSL_CMT_DRIVER_VERSION	39
7.6	Enumeration Type Documentation	39
7.6.1	cmt_mode_t	39
7.6.2	cmt_primary_clkdiv_t	39
7.6.3	cmt_second_clkdiv_t	39
7.6.4	cmt_infrared_output_polarity_t	40
7.6.5	cmt_infrared_output_state_t	40
7.6.6	_cmt_interrupt_enable	40
7.7	Function Documentation	40
7.7.1	CMT_GetDefaultConfig	40
7.7.2	CMT_Init	41
7.7.3	CMT_Deinit	41
7.7.4	CMT_SetMode	41
7.7.5	CMT_GetMode	41
7.7.6	CMT_GetCMTFrequency	42
7.7.7	CMT_SetCarrirGenerateCountOne	42
7.7.8	CMT_SetCarrirGenerateCountTwo	43
7.7.9	CMT_SetModulateMarkSpace	43
7.7.10	CMT_EnableExtendedSpace	43
7.7.11	CMT_SetIroState	44
7.7.12	CMT_EnableInterrupts	44
7.7.13	CMT_DisableInterrupts	44
7.7.14	CMT_GetStatusFlags	45

Contents

Section Number	Title	Page Number
Chapter	CRC: Cyclic Redundancy Check Driver	
8.1	Overview	47
8.2	CRC Driver Initialization and Configuration	47
8.3	CRC Write Data	47
8.4	CRC Get Checksum	47
8.5	Comments about API usage in RTOS	48
8.6	Data Structure Documentation	49
8.6.1	struct crc_config_t	49
8.7	Macro Definition Documentation	50
8.7.1	FSL_CRC_DRIVER_VERSION	50
8.7.2	CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT	50
8.8	Enumeration Type Documentation	50
8.8.1	crc_bits_t	50
8.8.2	crc_result_t	50
8.9	Function Documentation	50
8.9.1	CRC_Init	50
8.9.2	CRC_Deinit	51
8.9.3	CRC_GetDefaultConfig	51
8.9.4	CRC_WriteData	51
8.9.5	CRC_Get32bitResult	52
8.9.6	CRC_Get16bitResult	52
Chapter	DAC: Digital-to-Analog Converter Driver	
9.1	Overview	53
9.2	Typical use case	53
9.2.1	Working as a basic DAC without the hardware buffer feature	53
9.2.2	Working with the hardware buffer	53
9.3	Data Structure Documentation	55
9.3.1	struct dac_config_t	55
9.3.2	struct dac_buffer_config_t	55
9.4	Macro Definition Documentation	56
9.4.1	FSL_DAC_DRIVER_VERSION	56
9.5	Enumeration Type Documentation	56

Contents

Section Number	Title	Page Number
9.5.1	_dac_buffer_status_flags	56
9.5.2	_dac_buffer_interrupt_enable	56
9.5.3	dac_reference_voltage_source_t	56
9.5.4	dac_buffer_trigger_mode_t	57
9.5.5	dac_buffer_watermark_t	57
9.5.6	dac_buffer_work_mode_t	57
9.6	Function Documentation	57
9.6.1	DAC_Init	57
9.6.2	DAC_Deinit	57
9.6.3	DAC_GetDefaultConfig	58
9.6.4	DAC_Enable	58
9.6.5	DAC_EnableBuffer	58
9.6.6	DAC_SetBufferConfig	58
9.6.7	DAC_GetDefaultBufferConfig	59
9.6.8	DAC_EnableBufferDMA	59
9.6.9	DAC_SetBufferValue	59
9.6.10	DAC_DoSoftwareTriggerBuffer	60
9.6.11	DAC_GetBufferReadPointer	61
9.6.12	DAC_SetBufferReadPointer	61
9.6.13	DAC_EnableBufferInterrupts	61
9.6.14	DAC_DisableBufferInterrupts	61
9.6.15	DAC_GetBufferStatusFlags	62
9.6.16	DAC_ClearBufferStatusFlags	62

Chapter [DMAMUX: Direct Memory Access Multiplexer Driver](#)

10.1	Overview	63
10.2	Typical use case	63
10.2.1	DMAMUX Operation	63
10.3	Macro Definition Documentation	63
10.3.1	FSL_DMAMUX_DRIVER_VERSION	63
10.4	Function Documentation	64
10.4.1	DMAMUX_Init	64
10.4.2	DMAMUX_Deinit	65
10.4.3	DMAMUX_EnableChannel	65
10.4.4	DMAMUX_DisableChannel	65
10.4.5	DMAMUX_SetSource	66
10.4.6	DMAMUX_EnablePeriodTrigger	66
10.4.7	DMAMUX_DisablePeriodTrigger	66

Contents

Section Number	Title	Page Number
Chapter	DSPI: Serial Peripheral Interface Driver	
11.1	Overview	67
11.2	DSPI Driver	68
11.2.1	Overview	68
11.2.2	Typical use case	68
11.2.3	Data Structure Documentation	75
11.2.4	Macro Definition Documentation	82
11.2.5	Typedef Documentation	83
11.2.6	Enumeration Type Documentation	84
11.2.7	Function Documentation	88
11.2.8	Variable Documentation	106
11.3	DSPI DMA Driver	107
11.3.1	Overview	107
11.3.2	Data Structure Documentation	108
11.3.3	Macro Definition Documentation	111
11.3.4	Typedef Documentation	111
11.3.5	Function Documentation	112
11.4	DSPI eDMA Driver	117
11.4.1	Overview	117
11.4.2	Data Structure Documentation	118
11.4.3	Macro Definition Documentation	121
11.4.4	Typedef Documentation	121
11.4.5	Function Documentation	122
11.5	DSPI FreeRTOS Driver	127
11.5.1	Overview	127
11.5.2	Macro Definition Documentation	127
11.5.3	Function Documentation	127
Chapter	eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	
12.1	Overview	131
12.2	Typical use case	131
12.2.1	eDMA Operation	131
12.3	Data Structure Documentation	136
12.3.1	struct edma_config_t	136
12.3.2	struct edma_transfer_config_t	136
12.3.3	struct edma_channel_Preemption_config_t	137
12.3.4	struct edma_minor_offset_config_t	138
12.3.5	struct edma_tcd_t	138

Section Number	Title	Page Number
12.3.6	<code>struct edma_handle_t</code>	139
12.4	Macro Definition Documentation	140
12.4.1	<code>FSL_EDMA_DRIVER_VERSION</code>	140
12.5	Typedef Documentation	140
12.5.1	<code>edma_callback</code>	140
12.6	Enumeration Type Documentation	141
12.6.1	<code>edma_transfer_size_t</code>	141
12.6.2	<code>edma_modulo_t</code>	141
12.6.3	<code>edma_bandwidth_t</code>	142
12.6.4	<code>edma_channel_link_type_t</code>	142
12.6.5	<code>_edma_channel_status_flags</code>	142
12.6.6	<code>_edma_error_status_flags</code>	142
12.6.7	<code>edma_interrupt_enable_t</code>	143
12.6.8	<code>edma_transfer_type_t</code>	143
12.6.9	<code>_edma_transfer_status</code>	143
12.7	Function Documentation	143
12.7.1	<code>EDMA_Init</code>	143
12.7.2	<code>EDMA_Deinit</code>	144
12.7.3	<code>EDMA_InstallTCD</code>	144
12.7.4	<code>EDMA_GetDefaultConfig</code>	144
12.7.5	<code>EDMA_ResetChannel</code>	145
12.7.6	<code>EDMA_SetTransferConfig</code>	145
12.7.7	<code>EDMA_SetMinorOffsetConfig</code>	146
12.7.8	<code>EDMA_SetChannelPreemptionConfig</code>	146
12.7.9	<code>EDMA_SetChannelLink</code>	146
12.7.10	<code>EDMA_SetBandWidth</code>	148
12.7.11	<code>EDMA_SetModulo</code>	149
12.7.12	<code>EDMA_EnableAutoStopRequest</code>	149
12.7.13	<code>EDMA_EnableChannelInterrupts</code>	149
12.7.14	<code>EDMA_DisableChannelInterrupts</code>	150
12.7.15	<code>EDMA_TcdReset</code>	151
12.7.16	<code>EDMA_TcdSetTransferConfig</code>	151
12.7.17	<code>EDMA_TcdSetMinorOffsetConfig</code>	152
12.7.18	<code>EDMA_TcdSetChannelLink</code>	152
12.7.19	<code>EDMA_TcdSetBandWidth</code>	153
12.7.20	<code>EDMA_TcdSetModulo</code>	153
12.7.21	<code>EDMA_TcdEnableAutoStopRequest</code>	153
12.7.22	<code>EDMA_TcdEnableInterrupts</code>	154
12.7.23	<code>EDMA_TcdDisableInterrupts</code>	155
12.7.24	<code>EDMA_EnableChannelRequest</code>	155
12.7.25	<code>EDMA_DisableChannelRequest</code>	155

Section Number	Title	Page Number
12.7.26	EDMA_TriggerChannelStart	156
12.7.27	EDMA_GetRemainingMajorLoopCount	157
12.7.28	EDMA_GetErrorStatusFlags	157
12.7.29	EDMA_GetChannelStatusFlags	158
12.7.30	EDMA_ClearChannelStatusFlags	158
12.7.31	EDMA_CreateHandle	158
12.7.32	EDMA_InstallTCDMemory	159
12.7.33	EDMA_SetCallback	159
12.7.34	EDMA_PreparesTransfer	159
12.7.35	EDMA_SubmitTransfer	160
12.7.36	EDMA_StartTransfer	161
12.7.37	EDMA_StopTransfer	162
12.7.38	EDMA_AbortTransfer	162
12.7.39	EDMA_GetUnusedTCDNumber	162
12.7.40	EDMA_GetNextTCDAddress	162
12.7.41	EDMA_HandleIRQ	163
Chapter	ENET: Ethernet MAC Driver	
13.1	Overview	165
13.2	Typical use case	166
13.2.1	ENET Initialization, receive, and transmit operations	166
13.3	Data Structure Documentation	174
13.3.1	struct enet_rx_bd_struct_t	174
13.3.2	struct enet_tx_bd_struct_t	175
13.3.3	struct enet_data_error_stats_t	176
13.3.4	struct enet_buffer_config_t	177
13.3.5	struct enet_ptp_time_t	178
13.3.6	struct enet_ptp_time_data_t	179
13.3.7	struct enet_ptp_time_data_ring_t	179
13.3.8	struct enet_ptp_config_t	180
13.3.9	struct enet_config_t	180
13.3.10	struct _enet_handle	182
13.4	Macro Definition Documentation	184
13.4.1	FSL_ENET_DRIVER_VERSION	184
13.4.2	FSL_FEATURE_ENET_QUEUE	186
13.4.3	ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK	186
13.4.4	ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK	186
13.4.5	ENET_BUFFDESCRIPTOR_RX_WRAP_MASK	186
13.4.6	ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask	186
13.4.7	ENET_BUFFDESCRIPTOR_RX_LAST_MASK	186
13.4.8	ENET_BUFFDESCRIPTOR_RX_MISS_MASK	186

Contents

Section Number	Title	Page Number
13.4.9	ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK	186
13.4.10	ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK	186
13.4.11	ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK	186
13.4.12	ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK	186
13.4.13	ENET_BUFFDESCRIPTOR_RX_CRC_MASK	186
13.4.14	ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK	186
13.4.15	ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK	186
13.4.16	ENET_BUFFDESCRIPTOR_TX_READY_MASK	186
13.4.17	ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK	186
13.4.18	ENET_BUFFDESCRIPTOR_TX_WRAP_MASK	186
13.4.19	ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK	186
13.4.20	ENET_BUFFDESCRIPTOR_TX_LAST_MASK	186
13.4.21	ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK	186
13.4.22	ENET_BUFFDESCRIPTOR_RX_IPV4_MASK	186
13.4.23	ENET_BUFFDESCRIPTOR_RX_IPV6_MASK	186
13.4.24	ENET_BUFFDESCRIPTOR_RX_VLAN_MASK	186
13.4.25	ENET_BUFFDESCRIPTOR_RX_PROTOCOLCHECKSUM_MASK	186
13.4.26	ENET_BUFFDESCRIPTOR_RX_IPHEADCHECKSUM_MASK	186
13.4.27	ENET_BUFFDESCRIPTOR_RX_INTERRUPT_MASK	186
13.4.28	ENET_BUFFDESCRIPTOR_RX_UNICAST_MASK	186
13.4.29	ENET_BUFFDESCRIPTOR_RX_COLLISION_MASK	186
13.4.30	ENET_BUFFDESCRIPTOR_RX_PHYERR_MASK	186
13.4.31	ENET_BUFFDESCRIPTOR_RX_MACERR_MASK	186
13.4.32	ENET_BUFFDESCRIPTOR_TX_ERR_MASK	186
13.4.33	ENET_BUFFDESCRIPTOR_TX_UNDERFLOWERR_MASK	186
13.4.34	ENET_BUFFDESCRIPTOR_TX_EXCCOLLISIONERR_MASK	186
13.4.35	ENET_BUFFDESCRIPTOR_TX_FRAMEERR_MASK	186
13.4.36	ENET_BUFFDESCRIPTOR_TX_LATECOLLISIONERR_MASK	186
13.4.37	ENET_BUFFDESCRIPTOR_TX_OVERFLOWERR_MASK	186
13.4.38	ENET_BUFFDESCRIPTOR_TX_TIMESTAMPERR_MASK	186
13.4.39	ENET_BUFFDESCRIPTOR_TX_INTERRUPT_MASK	186
13.4.40	ENET_BUFFDESCRIPTOR_TX_TIMESTAMP_MASK	186
13.4.41	ENET_BUFFDESCRIPTOR_RX_ERR_MASK	186
13.4.42	ENET_FRAME_MAX_FRAMELEN	187
13.4.43	ENET_FIFO_MIN_RX_FULL	187
13.4.44	ENET_RX_MIN_BUFFERSIZE	187
13.5	Typedef Documentation	187
13.5.1	enet_callback_t	187
13.6	Enumeration Type Documentation	187
13.6.1	_enet_status	187
13.6.2	enet_mii_mode_t	187
13.6.3	enet_mii_speed_t	187
13.6.4	enet_mii_duplex_t	188

Contents

Section Number	Title	Page Number
13.6.5	enet_mii_write_t	188
13.6.6	enet_mii_read_t	188
13.6.7	enet_special_control_flag_t	188
13.6.8	enet_interrupt_enable_t	189
13.6.9	enet_event_t	189
13.6.10	enet_tx_accelerator_t	190
13.6.11	enet_rx_accelerator_t	190
13.6.12	enet_ptp_event_type_t	190
13.6.13	enet_ptp_timer_channel_t	190
13.6.14	enet_ptp_timer_channel_mode_t	190
13.7	Function Documentation	191
13.7.1	ENET_GetDefaultConfig	191
13.7.2	ENET_Init	191
13.7.3	ENET_Deinit	192
13.7.4	ENET_Reset	192
13.7.5	ENET_SetMII	193
13.7.6	ENET_SetSMI	193
13.7.7	ENET_GetSMI	193
13.7.8	ENET_ReadSMIData	194
13.7.9	ENET_StartSMIRead	194
13.7.10	ENET_StartSMIWrite	194
13.7.11	ENET_SetMacAddr	195
13.7.12	ENET_GetMacAddr	195
13.7.13	ENET_AddMulticastGroup	195
13.7.14	ENET_LeaveMulticastGroup	195
13.7.15	ENET_ActiveRead	196
13.7.16	ENET_EnableSleepMode	196
13.7.17	ENET_GetAccelFunction	196
13.7.18	ENET_EnableInterrupts	197
13.7.19	ENET_DisableInterrupts	197
13.7.20	ENET_GetInterruptStatus	198
13.7.21	ENET_ClearInterruptStatus	198
13.7.22	ENET_SetCallback	198
13.7.23	ENET_GetRxErrBeforeReadFrame	199
13.7.24	ENET_GetTxErrAfterSendFrame	199
13.7.25	ENET_GetRxFrameSize	200
13.7.26	ENET_ReadFrame	200
13.7.27	ENET_SendFrame	201
13.7.28	ENET_TransmitIRQHandler	202
13.7.29	ENET_ReceiveIRQHandler	202
13.7.30	ENET_ErrorIRQHandler	202
13.7.31	ENET_CommonFrame0IRQHandler	203
13.7.32	ENET_Ptp1588Configure	203
13.7.33	ENET_Ptp1588StartTimer	203

Section Number	Title	Page Number
13.7.34	ENET_Ptp1588StopTimer	204
13.7.35	ENET_Ptp1588AdjustTimer	204
13.7.36	ENET_Ptp1588SetChannelMode	204
13.7.37	ENET_Ptp1588SetChannelCmpValue	205
13.7.38	ENET_Ptp1588GetChannelStatus	205
13.7.39	ENET_Ptp1588ClearChannelStatus	205
13.7.40	ENET_Ptp1588GetTimer	206
13.7.41	ENET_Ptp1588SetTimer	206
13.7.42	ENET_Ptp1588TimerIRQHandler	206
13.7.43	ENET_GetRxFrameTime	206
13.7.44	ENET_GetTxFrameTime	207

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
14.6.5	EWM_DisableInterrupts	212
14.6.6	EWM_GetStatusFlags	212
14.6.7	EWM_Refresh	213

Chapter C90TFS Flash Driver

Chapter FlexBus: External Bus Interface Driver

16.1	Overview	217
16.2	FlexBus functional operation	217

Section Number	Title	Page Number
16.3	Typical use case and example	217
16.4	Data Structure Documentation	219
16.4.1	struct flexbus_config_t	219
16.5	Macro Definition Documentation	220
16.5.1	FSL_FLEXBUS_DRIVER_VERSION	220
16.6	Enumeration Type Documentation	220
16.6.1	flexbus_port_size_t	220
16.6.2	flexbus_write_address_hold_t	220
16.6.3	flexbus_read_address_hold_t	220
16.6.4	flexbus_address_setup_t	221
16.6.5	flexbus_bytelane_shift_t	221
16.6.6	flexbus_multiplex_group1_t	221
16.6.7	flexbus_multiplex_group2_t	221
16.6.8	flexbus_multiplex_group3_t	222
16.6.9	flexbus_multiplex_group4_t	222
16.6.10	flexbus_multiplex_group5_t	222
16.7	Function Documentation	222
16.7.1	FLEXBUS_Init	222
16.7.2	FLEXBUS_Deinit	223
16.7.3	FLEXBUS_GetDefaultConfig	223

Chapter **FlexCAN: Flex Controller Area Network Driver**

17.1	Overview	225
17.2	FlexCAN Driver	226
17.2.1	Overview	226
17.2.2	Typical use case	226
17.2.3	Data Structure Documentation	233
17.2.4	Macro Definition Documentation	237
17.2.5	Typedef Documentation	242
17.2.6	Enumeration Type Documentation	242
17.2.7	Function Documentation	245
17.3	FlexCAN eDMA Driver	260
17.3.1	Overview	260
17.3.2	Data Structure Documentation	260
17.3.3	Macro Definition Documentation	261
17.3.4	Typedef Documentation	261
17.3.5	Function Documentation	261

Contents

Section Number	Title	Page Number
Chapter	FTM: FlexTimer Driver	
18.1	Overview	263
18.2	Function groups	263
18.2.1	Initialization and deinitialization	263
18.2.2	PWM Operations	263
18.2.3	Input capture operations	263
18.2.4	Output compare operations	264
18.2.5	Quad decode	264
18.2.6	Fault operation	264
18.3	Register Update	264
18.4	Typical use case	265
18.4.1	PWM output	265
18.5	Data Structure Documentation	271
18.5.1	struct ftm_chnl_pwm_signal_param_t	271
18.5.2	struct ftm_chnl_pwm_config_param_t	272
18.5.3	struct ftm_dual_edge_capture_param_t	272
18.5.4	struct ftm_phase_params_t	272
18.5.5	struct ftm_fault_param_t	273
18.5.6	struct ftm_config_t	273
18.6	Macro Definition Documentation	274
18.6.1	FSL_FTM_DRIVER_VERSION	274
18.7	Enumeration Type Documentation	274
18.7.1	ftm_chnl_t	274
18.7.2	ftm_fault_input_t	275
18.7.3	ftm_pwm_mode_t	275
18.7.4	ftm_pwm_level_select_t	275
18.7.5	ftm_output_compare_mode_t	275
18.7.6	ftm_input_capture_edge_t	275
18.7.7	ftm_dual_edge_capture_mode_t	276
18.7.8	ftm_quad_decode_mode_t	276
18.7.9	ftm_phase_polarity_t	276
18.7.10	ftm_deadtime_prescale_t	276
18.7.11	ftm_clock_source_t	276
18.7.12	ftm_clock_prescale_t	277
18.7.13	ftm_bdm_mode_t	277
18.7.14	ftm_fault_mode_t	277
18.7.15	ftm_external_trigger_t	277
18.7.16	ftm_pwm_sync_method_t	278
18.7.17	ftm_reload_point_t	278

Section Number	Title	Page Number
18.7.18	<code>ftm_interrupt_enable_t</code>	279
18.7.19	<code>ftm_status_flags_t</code>	279
18.7.20	<code>_ftm_quad_decoder_flags</code>	280
18.8	Function Documentation	280
18.8.1	<code>FTM_Init</code>	280
18.8.2	<code>FTM_Deinit</code>	280
18.8.3	<code>FTM_GetDefaultConfig</code>	280
18.8.4	<code>FTM_SetupPwm</code>	281
18.8.5	<code>FTM_UpdatePwmDutyCycle</code>	281
18.8.6	<code>FTM_UpdateChnlEdgeLevelSelect</code>	282
18.8.7	<code>FTM_SetupPwmMode</code>	282
18.8.8	<code>FTM_SetupInputCapture</code>	283
18.8.9	<code>FTM_SetupOutputCompare</code>	283
18.8.10	<code>FTM_SetupDualEdgeCapture</code>	283
18.8.11	<code>FTM_SetupFault</code>	284
18.8.12	<code>FTM_EnableInterrupts</code>	284
18.8.13	<code>FTM_DisableInterrupts</code>	284
18.8.14	<code>FTM_GetEnabledInterrupts</code>	285
18.8.15	<code>FTM_GetStatusFlags</code>	286
18.8.16	<code>FTM_ClearStatusFlags</code>	286
18.8.17	<code>FTM_SetTimerPeriod</code>	286
18.8.18	<code>FTM_GetCurrentTimerCount</code>	287
18.8.19	<code>FTM_StartTimer</code>	287
18.8.20	<code>FTM_StopTimer</code>	287
18.8.21	<code>FTM_SetSoftwareCtrlEnable</code>	287
18.8.22	<code>FTM_SetSoftwareCtrlVal</code>	288
18.8.23	<code>FTM_SetGlobalTimeBaseOutputEnable</code>	288
18.8.24	<code>FTM_SetOutputMask</code>	288
18.8.25	<code>FTM_SetFaultControlEnable</code>	289
18.8.26	<code>FTM_SetDeadTimeEnable</code>	290
18.8.27	<code>FTM_SetComplementaryEnable</code>	290
18.8.28	<code>FTM_SetInvertEnable</code>	290
18.8.29	<code>FTM_SetupQuadDecode</code>	291
18.8.30	<code>FTM_GetQuadDecoderFlags</code>	291
18.8.31	<code>FTM_SetQuadDecoderModuloValue</code>	291
18.8.32	<code>FTM_GetQuadDecoderCounterValue</code>	292
18.8.33	<code>FTM_ClearQuadDecoderCounterValue</code>	292
18.8.34	<code>FTM_SetSoftwareTrigger</code>	292
18.8.35	<code>FTM_SetWriteProtection</code>	292
19.1	Overview	295

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

19.1	Overview	295
-------------	---------------------------	------------

Contents

Section Number	Title	Page Number
19.2	Data Structure Documentation	295
19.2.1	struct gpio_pin_config_t	295
19.3	Macro Definition Documentation	296
19.3.1	FSL_GPIO_DRIVER_VERSION	296
19.4	Enumeration Type Documentation	296
19.4.1	gpio_pin_direction_t	296
19.5	GPIO Driver	297
19.5.1	Overview	297
19.5.2	Typical use case	297
19.5.3	Function Documentation	298
19.6	FGPIO Driver	301
19.6.1	Typical use case	301
Chapter I2C: Inter-Integrated Circuit Driver		
20.1	Overview	303
20.2	I2C Driver	304
20.2.1	Overview	304
20.2.2	Typical use case	304
20.2.3	Data Structure Documentation	309
20.2.4	Macro Definition Documentation	313
20.2.5	Typedef Documentation	313
20.2.6	Enumeration Type Documentation	313
20.2.7	Function Documentation	315
20.2.8	Variable Documentation	329
20.3	I2C eDMA Driver	330
20.3.1	Overview	330
20.3.2	Data Structure Documentation	330
20.3.3	Macro Definition Documentation	331
20.3.4	Typedef Documentation	331
20.3.5	Function Documentation	331
20.4	I2C DMA Driver	334
20.4.1	Overview	334
20.4.2	Data Structure Documentation	334
20.4.3	Macro Definition Documentation	335
20.4.4	Typedef Documentation	335
20.4.5	Function Documentation	335
20.5	I2C FreeRTOS Driver	338

Contents

Section Number	Title	Page Number
20.5.1	Overview	338
20.5.2	Macro Definition Documentation	338
20.5.3	Function Documentation	338
Chapter	LLWU: Low-Leakage Wakeup Unit Driver	
21.1	Overview	341
21.2	External wakeup pins configurations	341
21.3	Internal wakeup modules configurations	341
21.4	Digital pin filter for external wakeup pin configurations	341
21.5	Data Structure Documentation	342
21.5.1	struct llwu_external_pin_filter_mode_t	342
21.6	Macro Definition Documentation	342
21.6.1	FSL_LLWU_DRIVER_VERSION	342
21.7	Enumeration Type Documentation	343
21.7.1	llwu_external_pin_mode_t	343
21.7.2	llwu_pin_filter_mode_t	343
21.8	Function Documentation	343
21.8.1	LLWU_SetExternalWakeUpPinMode	343
21.8.2	LLWU_GetExternalWakeUpPinFlag	343
21.8.3	LLWU_ClearExternalWakeUpPinFlag	344
21.8.4	LLWU_EnableInternalModuleInterruptWakup	344
21.8.5	LLWU_GetInternalWakeUpModuleFlag	344
21.8.6	LLWU_SetPinFilterMode	345
21.8.7	LLWU_GetPinFilterFlag	345
21.8.8	LLWU_ClearPinFilterFlag	345
21.8.9	LLWU_SetResetPinMode	346
Chapter	LPTMR: Low-Power Timer	
22.1	Overview	347
22.2	Function groups	347
22.2.1	Initialization and deinitialization	347
22.2.2	Timer period Operations	347
22.2.3	Start and Stop timer operations	347
22.2.4	Status	348
22.2.5	Interrupt	348

Section Number	Title	Page Number
22.3	Typical use case	348
22.3.1	LPTMR tick example	348
22.4	Data Structure Documentation	350
22.4.1	struct lptmr_config_t	350
22.5	Enumeration Type Documentation	351
22.5.1	lptmr_pin_select_t	351
22.5.2	lptmr_pin_polarity_t	351
22.5.3	lptmr_timer_mode_t	351
22.5.4	lptmr_prescaler_glitch_value_t	351
22.5.5	lptmr_prescaler_clock_select_t	352
22.5.6	lptmr_interrupt_enable_t	352
22.5.7	lptmr_status_flags_t	352
22.6	Function Documentation	352
22.6.1	LPTMR_Init	352
22.6.2	LPTMR_Deinit	353
22.6.3	LPTMR_GetDefaultConfig	353
22.6.4	LPTMR_EnableInterrupts	353
22.6.5	LPTMR_DisableInterrupts	353
22.6.6	LPTMR_GetEnabledInterrupts	354
22.6.7	LPTMR_GetStatusFlags	354
22.6.8	LPTMR_ClearStatusFlags	354
22.6.9	LPTMR_SetTimerPeriod	355
22.6.10	LPTMR_GetCurrentTimerCount	355
22.6.11	LPTMR_StartTimer	355
22.6.12	LPTMR_StopTimer	356

Chapter PDB: Programmable Delay Block

23.1	Overview	357
23.2	Typical use case	357
23.2.1	Working as basic PDB counter with a PDB interrupt.	357
23.2.2	Working with an additional trigger. The ADC trigger is used as an example.	357
23.3	Data Structure Documentation	361
23.3.1	struct pdb_config_t	361
23.3.2	struct pdb_adc_pretrigger_config_t	362
23.3.3	struct pdb_dac_trigger_config_t	362
23.4	Macro Definition Documentation	363
23.4.1	FSL_PDB_DRIVER_VERSION	363
23.5	Enumeration Type Documentation	363

Contents

Section Number	Title	Page Number
23.5.1	<code>_pdb_status_flags</code>	363
23.5.2	<code>_pdb_adc_pretrigger_flags</code>	363
23.5.3	<code>_pdb_interrupt_enable</code>	363
23.5.4	<code>pdb_load_value_mode_t</code>	363
23.5.5	<code>pdb_prescaler_divider_t</code>	364
23.5.6	<code>pdb_divider_multiplication_factor_t</code>	364
23.5.7	<code>pdb_trigger_input_source_t</code>	364
23.5.8	<code>pdb_adc_trigger_channel_t</code>	365
23.5.9	<code>pdb_adc_pretrigger_t</code>	365
23.5.10	<code>pdb_dac_trigger_channel_t</code>	366
23.5.11	<code>pdb_pulse_out_trigger_channel_t</code>	366
23.5.12	<code>pdb_pulse_out_channel_mask_t</code>	366
23.6	Function Documentation	367
23.6.1	<code>PDB_Init</code>	367
23.6.2	<code>PDB_Deinit</code>	367
23.6.3	<code>PDB_GetDefaultConfig</code>	367
23.6.4	<code>PDB_Enable</code>	367
23.6.5	<code>PDB_DoSoftwareTrigger</code>	368
23.6.6	<code>PDB_DoLoadValues</code>	368
23.6.7	<code>PDB_EnableDMA</code>	368
23.6.8	<code>PDB_EnableInterrupts</code>	368
23.6.9	<code>PDB_DisableInterrupts</code>	369
23.6.10	<code>PDB_GetStatusFlags</code>	369
23.6.11	<code>PDB_ClearStatusFlags</code>	369
23.6.12	<code>PDB_SetModulusValue</code>	369
23.6.13	<code>PDB_GetCounterValue</code>	370
23.6.14	<code>PDB_SetCounterDelayValue</code>	370
23.6.15	<code>PDB_SetADCPreTriggerConfig</code>	370
23.6.16	<code>PDB_SetADCPreTriggerDelayValue</code>	371
23.6.17	<code>PDB_GetADCPreTriggerStatusFlags</code>	371
23.6.18	<code>PDB_ClearADCPreTriggerStatusFlags</code>	371
23.6.19	<code>PDB_SetDACTriggerConfig</code>	372
23.6.20	<code>PDB_SetDACTriggerIntervalValue</code>	372
23.6.21	<code>PDB_EnablePulseOutTrigger</code>	372
23.6.22	<code>PDB_SetPulseOutTriggerDelayValue</code>	373

Chapter **PIT: Periodic Interrupt Timer**

24.1	Overview	375
24.2	Function groups	375
24.2.1	Initialization and deinitialization	375
24.2.2	Timer period Operations	375
24.2.3	Start and Stop timer operations	375

Contents

Section Number	Title	Page Number
24.2.4	Status	376
24.2.5	Interrupt	376
24.3	Typical use case	376
24.3.1	PIT tick example	376
24.4	Data Structure Documentation	377
24.4.1	struct pit_config_t	377
24.5	Enumeration Type Documentation	377
24.5.1	pit_chnl_t	377
24.5.2	pit_interrupt_enable_t	378
24.5.3	pit_status_flags_t	378
24.6	Function Documentation	378
24.6.1	PIT_Init	378
24.6.2	PIT_Deinit	378
24.6.3	PIT_GetDefaultConfig	379
24.6.4	PIT_SetTimerChainMode	379
24.6.5	PIT_EnableInterrupts	379
24.6.6	PIT_DisableInterrupts	380
24.6.7	PIT_GetEnabledInterrupts	380
24.6.8	PIT_GetStatusFlags	380
24.6.9	PIT_ClearStatusFlags	381
24.6.10	PIT_SetTimerPeriod	381
24.6.11	PIT_GetCurrentTimerCount	382
24.6.12	PIT_StartTimer	382
24.6.13	PIT_StopTimer	382

Chapter **PMC: Power Management Controller**

25.1	Overview	385
25.2	Data Structure Documentation	386
25.2.1	struct pmc_low_volt_detect_config_t	386
25.2.2	struct pmc_low_volt_warning_config_t	386
25.2.3	struct pmc_bandgap_buffer_config_t	386
25.3	Macro Definition Documentation	387
25.3.1	FSL_PMC_DRIVER_VERSION	387
25.4	Enumeration Type Documentation	387
25.4.1	pmc_low_volt_detect_volt_select_t	387
25.4.2	pmc_low_volt_warning_volt_select_t	387
25.5	Function Documentation	387

Contents

Section Number	Title	Page Number
25.5.1	PMC_ConfigureLowVoltDetect	387
25.5.2	PMC_GetLowVoltDetectFlag	388
25.5.3	PMC_ClearLowVoltDetectFlag	388
25.5.4	PMC_ConfigureLowVoltWarning	388
25.5.5	PMC_GetLowVoltWarningFlag	389
25.5.6	PMC_ClearLowVoltWarningFlag	389
25.5.7	PMC_ConfigureBandgapBuffer	389
25.5.8	PMC_GetPeriphIOIsolationFlag	390
25.5.9	PMC_ClearPeriphIOIsolationFlag	390
25.5.10	PMC_IsRegulatorInRunRegulation	390
 Chapter PORT: Port Control and Interrupts		
26.1	Overview	393
26.2	Data Structure Documentation	395
26.2.1	struct port_digital_filter_config_t	395
26.2.2	struct port_pin_config_t	395
26.3	Macro Definition Documentation	396
26.3.1	FSL_PORT_DRIVER_VERSION	396
26.4	Enumeration Type Documentation	396
26.4.1	_port_pull	396
26.4.2	_port_slew_rate	396
26.4.3	_port_open_drain_enable	396
26.4.4	_port_passive_filter_enable	396
26.4.5	_port_drive_strength	396
26.4.6	_port_lock_register	397
26.4.7	port_mux_t	397
26.4.8	port_interrupt_t	397
26.4.9	port_digital_filter_clock_source_t	398
26.5	Function Documentation	398
26.5.1	PORT_SetPinConfig	398
26.5.2	PORT_SetMultiplePinsConfig	398
26.5.3	PORT_SetPinMux	399
26.5.4	PORT_EnablePinsDigitalFilter	399
26.5.5	PORT_SetDigitalFilterConfig	400
26.5.6	PORT_SetPinInterruptConfig	401
26.5.7	PORT_SetPinDriveStrength	401
26.5.8	PORT_GetPinsInterruptFlags	402
26.5.9	PORT_ClearPinsInterruptFlags	402

Contents

Section Number	Title	Page Number
Chapter	RCM: Reset Control Module Driver	
27.1	Overview	403
27.2	Data Structure Documentation	404
27.2.1	struct rcm_reset_pin_filter_config_t	404
27.3	Macro Definition Documentation	404
27.3.1	FSL_RCM_DRIVER_VERSION	404
27.4	Enumeration Type Documentation	404
27.4.1	rcm_reset_source_t	404
27.4.2	rcm_run_wait_filter_mode_t	405
27.5	Function Documentation	405
27.5.1	RCM_GetPreviousResetSources	405
27.5.2	RCM_ConfigureResetPinFilter	405
27.5.3	RCM_GetEasyPortModePinStatus	406
Chapter	RNGA: Random Number Generator Accelerator Driver	
28.1	Overview	407
28.2	RNGA Initialization	407
28.3	Get random data from RNGA	407
28.4	RNGA Set/Get Working Mode	407
28.5	Seed RNGA	407
28.6	Macro Definition Documentation	408
28.6.1	FSL_RNGA_DRIVER_VERSION	408
28.7	Enumeration Type Documentation	409
28.7.1	rnga_mode_t	409
28.8	Function Documentation	409
28.8.1	RNGA_Init	409
28.8.2	RNGA_Deinit	409
28.8.3	RNGA_GetRandomData	409
28.8.4	RNGA_Seed	410
28.8.5	RNGA_SetMode	410
28.8.6	RNGA_GetMode	410

Section Number	Title	Page Number
Chapter	RTC: Real Time Clock	
29.1	Overview	411
29.2	Function groups	411
29.2.1	Initialization and deinitialization	411
29.2.2	Set & Get Datetime	411
29.2.3	Set & Get Alarm	411
29.2.4	Start & Stop timer	411
29.2.5	Status	412
29.2.6	Interrupt	412
29.2.7	RTC Oscillator	412
29.2.8	Monotonic Counter	412
29.3	Typical use case	412
29.3.1	RTC tick example	412
29.4	Data Structure Documentation	414
29.4.1	struct rtc_datetime_t	414
29.4.2	struct rtc_config_t	415
29.5	Enumeration Type Documentation	415
29.5.1	rtc_interrupt_enable_t	415
29.5.2	rtc_status_flags_t	416
29.5.3	rtc_osc_cap_load_t	416
29.6	Function Documentation	416
29.6.1	RTC_Init	416
29.6.2	RTC_Deinit	416
29.6.3	RTC_GetDefaultConfig	417
29.6.4	RTC_SetDatetime	417
29.6.5	RTC_GetDatetime	417
29.6.6	RTC_SetAlarm	418
29.6.7	RTC_GetAlarm	418
29.6.8	RTC_EnableInterrupts	418
29.6.9	RTC_DisableInterrupts	419
29.6.10	RTC_GetEnabledInterrupts	419
29.6.11	RTC_GetStatusFlags	419
29.6.12	RTC_ClearStatusFlags	419
29.6.13	RTC_SetClockSource	420
29.6.14	RTC_StartTimer	420
29.6.15	RTC_StopTimer	420
29.6.16	RTC_SetOscCapLoad	420
29.6.17	RTC_Reset	421

Contents

Section Number	Title	Page Number
Chapter	SAI: Serial Audio Interface	
30.1	Overview	423
30.2	Typical use case	423
30.2.1	SAI Send/receive using an interrupt method	423
30.2.2	SAI Send/receive using a DMA method	423
30.3	Data Structure Documentation	429
30.3.1	struct sai_config_t	429
30.3.2	struct sai_transfer_format_t	430
30.3.3	struct sai_transfer_t	430
30.3.4	struct _sai_handle	431
30.4	Macro Definition Documentation	431
30.4.1	SAI_XFER_QUEUE_SIZE	431
30.5	Enumeration Type Documentation	431
30.5.1	_sai_status_t	431
30.5.2	_sai_channel_mask	432
30.5.3	sai_protocol_t	432
30.5.4	sai_master_slave_t	432
30.5.5	sai_mono_stereo_t	432
30.5.6	sai_data_order_t	433
30.5.7	sai_clock_polarity_t	433
30.5.8	sai_sync_mode_t	433
30.5.9	sai_mclk_source_t	433
30.5.10	sai_bclk_source_t	434
30.5.11	_sai_interrupt_enable_t	434
30.5.12	_sai_dma_enable_t	434
30.5.13	_sai_flags	434
30.5.14	sai_reset_type_t	435
30.5.15	sai_sample_rate_t	435
30.5.16	sai_word_width_t	435
30.6	Function Documentation	435
30.6.1	SAI_TxInit	435
30.6.2	SAI_RxInit	436
30.6.3	SAI_TxGetDefaultConfig	436
30.6.4	SAI_RxGetDefaultConfig	436
30.6.5	SAI_Deinit	437
30.6.6	SAI_TxReset	437
30.6.7	SAI_RxReset	437
30.6.8	SAI_TxEnable	437
30.6.9	SAI_RxEnable	438
30.6.10	SAI_TxGetStatusFlag	438

Contents

Section Number	Title	Page Number
30.6.11	SAI_TxClearStatusFlags	438
30.6.12	SAI_RxGetStatusFlag	438
30.6.13	SAI_RxClearStatusFlags	439
30.6.14	SAI_TxSoftwareReset	439
30.6.15	SAI_RxSoftwareReset	439
30.6.16	SAI_TxSetChannelFIFOMask	440
30.6.17	SAI_RxSetChannelFIFOMask	440
30.6.18	SAI_TxSetDataOrder	440
30.6.19	SAI_RxSetDataOrder	440
30.6.20	SAI_TxSetBitClockPolarity	441
30.6.21	SAI_RxSetBitClockPolarity	442
30.6.22	SAI_TxSetFrameSyncPolarity	442
30.6.23	SAI_RxSetFrameSyncPolarity	442
30.6.24	SAI_TxEnableInterrupts	442
30.6.25	SAI_RxEnableInterrupts	443
30.6.26	SAI_TxDisableInterrupts	444
30.6.27	SAI_RxDisableInterrupts	444
30.6.28	SAI_TxEnableDMA	445
30.6.29	SAI_RxEnableDMA	445
30.6.30	SAI_TxGetDataRegisterAddress	445
30.6.31	SAI_RxGetDataRegisterAddress	446
30.6.32	SAI_TxSetFormat	447
30.6.33	SAI_RxSetFormat	447
30.6.34	SAI_WriteBlocking	448
30.6.35	SAI_WriteMultiChannelBlocking	448
30.6.36	SAI_WriteData	448
30.6.37	SAI_ReadBlocking	449
30.6.38	SAI_ReadMultiChannelBlocking	449
30.6.39	SAI_ReadData	450
30.6.40	SAI_TransferTxCreateHandle	451
30.6.41	SAI_TransferRxCreateHandle	451
30.6.42	SAI_TransferTxSetFormat	452
30.6.43	SAI_TransferRxSetFormat	453
30.6.44	SAI_TransferSendNonBlocking	453
30.6.45	SAI_TransferReceiveNonBlocking	454
30.6.46	SAI_TransferGetSendCount	455
30.6.47	SAI_TransferGetReceiveCount	456
30.6.48	SAI_TransferAbortSend	456
30.6.49	SAI_TransferAbortReceive	457
30.6.50	SAI_TransferTerminateSend	457
30.6.51	SAI_TransferTerminateReceive	457
30.6.52	SAI_TransferTxHandleIRQ	458
30.6.53	SAI_TransferRxHandleIRQ	458
30.7	SAI DMA Driver	459

Section Number	Title	Page Number
30.7.1	Overview	459
30.7.2	Data Structure Documentation	460
30.7.3	Function Documentation	460
30.8	SAI EDMA Driver	466
30.8.1	Overview	466
30.8.2	Data Structure Documentation	467
30.8.3	Function Documentation	468
30.9	SAI SDMA Driver	475
30.9.1	Overview	475
30.9.2	Data Structure Documentation	476
30.9.3	Function Documentation	477
Chapter SDHC: Secure Digital Host Controller Driver		
31.1	Overview	483
31.2	Typical use case	483
31.2.1	SDHC Operation	483
31.3	Data Structure Documentation	491
31.3.1	struct sdhc_adma2_descriptor_t	491
31.3.2	struct sdhc_capability_t	491
31.3.3	struct sdhc_transfer_config_t	491
31.3.4	struct sdhc_boot_config_t	492
31.3.5	struct sdhc_config_t	492
31.3.6	struct sdhc_data_t	493
31.3.7	struct sdhc_command_t	493
31.3.8	struct sdhc_transfer_t	494
31.3.9	struct sdhc_transfer_callback_t	494
31.3.10	struct _sdhc_handle	494
31.3.11	struct sdhc_host_t	495
31.4	Macro Definition Documentation	495
31.4.1	FSL_SDHC_DRIVER_VERSION	495
31.5	Typedef Documentation	496
31.5.1	sdhc_adma1_descriptor_t	496
31.5.2	sdhc_transfer_function_t	496
31.6	Enumeration Type Documentation	496
31.6.1	_sdhc_status	496
31.6.2	_sdhc_capability_flag	496
31.6.3	_sdhc_wakeup_event	496
31.6.4	_sdhc_reset	497

Contents

Section Number	Title	Page Number
31.6.5	_sdhc_transfer_flag	497
31.6.6	_sdhc_present_status_flag	497
31.6.7	_sdhc_interrupt_status_flag	498
31.6.8	_sdhc_auto_command12_error_status_flag	498
31.6.9	_sdhc_adma_error_status_flag	499
31.6.10	sdhc_adma_error_state_t	499
31.6.11	_sdhc_force_event	499
31.6.12	sdhc_data_bus_width_t	500
31.6.13	sdhc_endian_mode_t	500
31.6.14	sdhc_dma_mode_t	500
31.6.15	_sdhc_sdio_control_flag	500
31.6.16	sdhc_boot_mode_t	501
31.6.17	sdhc_card_command_type_t	501
31.6.18	sdhc_card_response_type_t	501
31.6.19	_sdhc_adma1_descriptor_flag	501
31.6.20	_sdhc_adma2_descriptor_flag	502
31.7	Function Documentation	502
31.7.1	SDHC_Init	502
31.7.2	SDHC_Deinit	503
31.7.3	SDHC_Reset	503
31.7.4	SDHC_SetAdmaTableConfig	503
31.7.5	SDHC_EnableInterruptStatus	504
31.7.6	SDHC_DisableInterruptStatus	504
31.7.7	SDHC_EnableInterruptSignal	504
31.7.8	SDHC_DisableInterruptSignal	505
31.7.9	SDHC_GetInterruptStatusFlags	505
31.7.10	SDHC_ClearInterruptStatusFlags	505
31.7.11	SDHC_GetAutoCommand12ErrorStatusFlags	505
31.7.12	SDHC_GetAdmaErrorStatusFlags	506
31.7.13	SDHC_GetPresentStatusFlags	506
31.7.14	SDHC_GetCapability	506
31.7.15	SDHC_EnableSdClock	507
31.7.16	SDHC_SetSdClock	507
31.7.17	SDHC_SetCardActive	507
31.7.18	SDHC_SetDataBusWidth	508
31.7.19	SDHC_CardDetectByData3	508
31.7.20	SDHC_SetTransferConfig	508
31.7.21	SDHC_GetCommandResponse	509
31.7.22	SDHC_WriteData	509
31.7.23	SDHC_ReadData	509
31.7.24	SDHC_EnableWakeupEvent	510
31.7.25	SDHC_EnableCardDetectTest	510
31.7.26	SDHC_SetCardDetectTestLevel	510
31.7.27	SDHC_EnableSdioControl	511

Section Number	Contents	Page Number
	Title	
31.7.28	SDHC_SetContinueRequest	511
31.7.29	SDHC_SetMmcBootConfig	511
31.7.30	SDHC_SetForceEvent	512
31.7.31	SDHC_TransferBlocking	512
31.7.32	SDHC_TransferCreateHandle	513
31.7.33	SDHC_TransferNonBlocking	513
31.7.34	SDHC_TransferHandleIRQ	514
Chapter	SIM: System Integration Module Driver	
32.1	Overview	515
32.2	Data Structure Documentation	516
32.2.1	struct sim_uid_t	516
32.3	Enumeration Type Documentation	516
32.3.1	_sim_usb_volt_reg_enable_mode	516
32.3.2	_sim_flash_mode	516
32.4	Function Documentation	516
32.4.1	SIM_SetUsbVoltRegulatorEnableMode	516
32.4.2	SIM_GetUniqueId	518
32.4.3	SIM_SetFlashMode	518
Chapter	SMC: System Mode Controller Driver	
33.1	Overview	519
33.2	Typical use case	519
33.2.1	Enter wait or stop modes	519
33.3	Data Structure Documentation	521
33.3.1	struct smc_power_mode_vlls_config_t	521
33.4	Macro Definition Documentation	522
33.4.1	FSL_SMC_DRIVER_VERSION	522
33.5	Enumeration Type Documentation	522
33.5.1	smc_power_mode_protection_t	522
33.5.2	smc_power_state_t	522
33.5.3	smc_run_mode_t	522
33.5.4	smc_stop_mode_t	522
33.5.5	smc_stop_submode_t	523
33.5.6	smc_partial_stop_option_t	523
33.5.7	_smc_status	523

Contents

Section Number	Title	Page Number
33.6	Function Documentation	523
33.6.1	SMC_SetPowerModeProtection	523
33.6.2	SMC_GetPowerModeState	524
33.6.3	SMC_PreEnterStopModes	524
33.6.4	SMC_PostExitStopModes	524
33.6.5	SMC_PreEnterWaitModes	524
33.6.6	SMC_PostExitWaitModes	524
33.6.7	SMC_SetPowerModeRun	525
33.6.8	SMC_SetPowerModeWait	526
33.6.9	SMC_SetPowerModeStop	526
33.6.10	SMC_SetPowerModeVlpr	526
33.6.11	SMC_SetPowerModeVlpw	527
33.6.12	SMC_SetPowerModeVlps	527
33.6.13	SMC_SetPowerModeLls	527
33.6.14	SMC_SetPowerModeVlls	527
Chapter	UART: Universal Asynchronous Receiver/Transmitter Driver	
34.1	Overview	529
34.2	UART Driver	530
34.2.1	Overview	530
34.2.2	Typical use case	530
34.2.3	Data Structure Documentation	535
34.2.4	Macro Definition Documentation	538
34.2.5	Typedef Documentation	538
34.2.6	Enumeration Type Documentation	538
34.2.7	Function Documentation	540
34.3	UART DMA Driver	554
34.3.1	Overview	554
34.3.2	Data Structure Documentation	555
34.3.3	Macro Definition Documentation	556
34.3.4	Typedef Documentation	556
34.3.5	Function Documentation	556
34.4	UART eDMA Driver	560
34.4.1	Overview	560
34.4.2	Data Structure Documentation	561
34.4.3	Macro Definition Documentation	562
34.4.4	Typedef Documentation	562
34.4.5	Function Documentation	562
34.5	UART FreeRTOS Driver	566
34.5.1	Overview	566

Contents

Section Number	Title	Page Number
34.5.2	Data Structure Documentation	566
34.5.3	Macro Definition Documentation	567
34.5.4	Function Documentation	567
Chapter VREF: Voltage Reference Driver		
35.1	Overview	569
35.2	Typical use case and example	569
35.3	Data Structure Documentation	570
35.3.1	struct vref_config_t	570
35.4	Macro Definition Documentation	570
35.4.1	FSL_VREF_DRIVER_VERSION	570
35.5	Enumeration Type Documentation	570
35.5.1	vref_buffer_mode_t	570
35.6	Function Documentation	570
35.6.1	VREF_Init	570
35.6.2	VREF_Deinit	571
35.6.3	VREF_GetDefaultConfig	571
35.6.4	VREF_SetTrimVal	571
35.6.5	VREF_GetTrimVal	572
Chapter WDOG: Watchdog Timer Driver		
36.1	Overview	573
36.2	Typical use case	573
36.3	Data Structure Documentation	575
36.3.1	struct wdog_work_mode_t	575
36.3.2	struct wdog_config_t	575
36.3.3	struct wdog_test_config_t	576
36.4	Macro Definition Documentation	576
36.4.1	FSL_WDOG_DRIVER_VERSION	576
36.5	Enumeration Type Documentation	576
36.5.1	wdog_clock_source_t	576
36.5.2	wdog_clock_prescaler_t	576
36.5.3	wdog_test_mode_t	577
36.5.4	wdog_tested_byte_t	577
36.5.5	_wdog_interrupt_enable_t	577

Contents

Section Number	Title	Page Number
36.5.6	<code>_wdog_status_flags_t</code>	577
36.6	Function Documentation	577
36.6.1	WDOG_GetDefaultConfig	577
36.6.2	WDOG_Init	578
36.6.3	WDOG_Deinit	578
36.6.4	WDOG_SetTestModeConfig	579
36.6.5	WDOG_Enable	579
36.6.6	WDOG_Disable	579
36.6.7	WDOG_EnableInterrupts	580
36.6.8	WDOG_DisableInterrupts	580
36.6.9	WDOG_GetStatusFlags	580
36.6.10	WDOG_ClearStatusFlags	581
36.6.11	WDOG_SetTimeoutValue	581
36.6.12	WDOG_SetWindowValue	582
36.6.13	WDOG_Unlock	582
36.6.14	WDOG_Refresh	582
36.6.15	WDOG_GetResetCount	583
36.6.16	WDOG_ClearResetCount	584

Chapter Clock Driver

37.1	Overview	585
37.2	Multipurpose Clock Generator (MCG)	586
37.2.1	Function description	586
37.2.2	Typical use case	588
37.2.3	Code Configuration Option	591

Chapter DMA Manager

38.1	Overview	593
38.2	Function groups	593
38.2.1	DMAMGR Initialization and De-initialization	593
38.2.2	DMAMGR Operation	593
38.3	Typical use case	593
38.3.1	DMAMGR static channel allocation	593
38.3.2	DMAMGR dynamic channel allocation	593
38.4	Data Structure Documentation	594
38.4.1	struct dmamanager_handle_t	594
38.5	Macro Definition Documentation	595
38.5.1	DMAMGR_DYNAMIC_ALLOCATE	595

Section Number	Title	Page Number
38.6	Enumeration Type Documentation	595
38.6.1	_dma_manager_status	595
38.7	Function Documentation	595
38.7.1	DMAMGR_Init	595
38.7.2	DMAMGR_Deinit	596
38.7.3	DMAMGR_RequestChannel	596
38.7.4	DMAMGR_ReleaseChannel	597
38.7.5	DMAMGR_IsChannelOccupied	598
 Chapter Memory-Mapped Cryptographic Acceleration Unit (MMCAU)		
39.1	Overview	599
39.2	Purpose	599
39.3	Library Features	599
39.4	CAU and mmCAU software library overview	599
39.5	mmCAU software library usage	600
39.6	Function Documentation	602
39.6.1	cau_aes_set_key	602
39.6.2	cau_aes_encrypt	603
39.6.3	cau_aes_decrypt	603
39.6.4	cau_des_chk_parity	604
39.6.5	cau_des_encrypt	604
39.6.6	cau_des_decrypt	605
39.6.7	cau_md5_initialize_output	605
39.6.8	cau_md5_hash_n	606
39.6.9	cau_md5_update	607
39.6.10	cau_md5_hash	607
39.6.11	cau_sha1_initialize_output	608
39.6.12	cau_sha1_hash_n	608
39.6.13	cau_sha1_update	609
39.6.14	cau_sha1_hash	610
39.6.15	cau_sha256_initialize_output	610
39.6.16	cau_sha256_hash_n	611
39.6.17	cau_sha256_update	611
39.6.18	cau_sha256_hash	612
39.6.19	MMCAU_AES_SetKey	613
39.6.20	MMCAU_AES_EncryptEcb	613
39.6.21	MMCAU_AES_DecryptEcb	614
39.6.22	MMCAU_DES_ChkParity	615
39.6.23	MMCAU_DES_EncryptEcb	615

Contents

Section Number	Title	Page Number
39.6.24	MMCAU_DES_DecryptEcb	616
39.6.25	MMCAU_MD5_InitializeOutput	617
39.6.26	MMCAU_MD5_HashN	617
39.6.27	MMCAU_MD5_Update	618
39.6.28	MMCAU_SHA1_InitializeOutput	619
39.6.29	MMCAU_SHA1_HashN	619
39.6.30	MMCAU_SHA1_Update	620
39.6.31	MMCAU_SHA256_InitializeOutput	621
39.6.32	MMCAU_SHA256_HashN	621
39.6.33	MMCAU_SHA256_Update	622

Chapter Secure Digital Card/Embedded MultiMedia Card (CARD)

40.1	Overview	625
40.2	SD CARD Operation	625
40.3	MMC CARD Operation	627
40.4	SDIO CARD Operation	628
40.5	Data Structure Documentation	630
40.5.1	struct sdmmchost_detect_card_t	630
40.5.2	struct sdmmchost_pwr_card_t	631
40.6	Enumeration Type Documentation	631
40.6.1	_sdmmchost_endian_mode	631
40.6.2	sdmmchost_detect_card_type_t	631
40.7	Function Documentation	631
40.7.1	SDMMCHOST_NotSupport	631
40.7.2	SDMMCHOST_WaitCardDetectStatus	632
40.7.3	SDMMCHOST_IsCardPresent	632
40.7.4	SDMMCHOST_Init	632
40.7.5	SDMMCHOST_Reset	633
40.7.6	SDMMCHOST_ErrorRecovery	633
40.7.7	SDMMCHOST_Deinit	633
40.7.8	SDMMCHOST_PowerOffCard	633
40.7.9	SDMMCHOST_PowerOnCard	634
40.7.10	SDMMCHOST_Delay	634

Chapter SPI based Secure Digital Card (SDSPI)

41.1	Overview	635
41.2	Data Structure Documentation	637

Section Number	Title	Page Number
41.2.1	<code>struct sdspi_host_t</code>	637
41.2.2	<code>struct sdspi_card_t</code>	637
41.3	Enumeration Type Documentation	638
41.3.1	<code>_sdspi_status</code>	638
41.3.2	<code>_sdspi_card_flag</code>	639
41.3.3	<code>_sdspi_response_type</code>	639
41.3.4	<code>_sdspi_cmd</code>	639
41.4	Function Documentation	639
41.4.1	<code>SDSPI_Init</code>	639
41.4.2	<code>SDSPI_Deinit</code>	640
41.4.3	<code>SDSPI_CheckReadOnly</code>	640
41.4.4	<code>SDSPI_ReadBlocks</code>	641
41.4.5	<code>SDSPI_WriteBlocks</code>	641
41.4.6	<code>SDSPI_SendCid</code>	642
41.4.7	<code>SDSPI_SendPreErase</code>	643
41.4.8	<code>SDSPI_EraseBlocks</code>	643
41.4.9	<code>SDSPI_SwitchToHighSpeed</code>	644
 Chapter Debug Console		
42.1	Overview	645
42.2	Function groups	645
42.2.1	Initialization	645
42.2.2	Advanced Feature	645
42.3	Typical use case	649
42.4	Macro Definition Documentation	651
42.4.1	<code>DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN</code>	651
42.4.2	<code>DEBUGCONSOLE_REDIRECT_TO_SDK</code>	652
42.4.3	<code>DEBUGCONSOLE_DISABLE</code>	652
42.4.4	<code>SDK_DEBUGCONSOLE</code>	652
42.4.5	<code>SDK_DEBUGCONSOLE_UART</code>	652
42.4.6	<code>PRINTF</code>	652
42.5	Function Documentation	652
42.5.1	<code>DbgConsole_Init</code>	652
42.5.2	<code>DbgConsole_Deinit</code>	653
42.5.3	<code>DbgConsole_Printf</code>	653
42.5.4	<code>DbgConsole_Putchar</code>	653
42.5.5	<code>DbgConsole_Scanf</code>	654
42.5.6	<code>DbgConsole_Getchar</code>	654
42.5.7	<code>DbgConsole_Flush</code>	654

Section Number	Contents	Page Number
	Title	
42.5.8	StrFormatPrintf	655
42.5.9	StrFormatScanf	655
42.6	Semihosting	656
42.6.1	Guide Semihosting for IAR	656
42.6.2	Guide Semihosting for Keil µVision	656
42.6.3	Guide Semihosting for MCUXpresso IDE	657
42.6.4	Guide Semihosting for ARMGCC	657
42.7	SWO	660
42.7.1	Guide SWO for SDK	660
42.7.2	Guide SWO for Keil µVision	661
42.7.3	Guide SWO for MCUXpresso IDE	662
42.7.4	Guide SWO for ARMGCC	662
Chapter Notification Framework		
43.1	Overview	663
43.2	Notifier Overview	663
43.3	Data Structure Documentation	665
43.3.1	struct notifier_notification_block_t	665
43.3.2	struct notifier_callback_config_t	666
43.3.3	struct notifier_handle_t	666
43.4	Typedef Documentation	667
43.4.1	notifier_user_config_t	667
43.4.2	notifier_user_function_t	667
43.4.3	notifier_callback_t	668
43.5	Enumeration Type Documentation	668
43.5.1	_notifier_status	668
43.5.2	notifier_policy_t	669
43.5.3	notifier_notification_type_t	669
43.5.4	notifier_callback_type_t	669
43.6	Function Documentation	670
43.6.1	NOTIFIER_CreateHandle	670
43.6.2	NOTIFIER_SwitchConfig	671
43.6.3	NOTIFIER_GetErrorCallbackIndex	672
Chapter Shell		
44.1	Overview	673

Contents

Section Number	Title	Page Number
44.2	Function groups	673
44.2.1	Initialization	673
44.2.2	Advanced Feature	673
44.2.3	Shell Operation	674
44.3	Data Structure Documentation	675
44.3.1	struct shell_command_t	675
44.4	Macro Definition Documentation	676
44.4.1	SHELL_NON_BLOCKING_MODE	676
44.4.2	SHELL_AUTO_COMPLETE	676
44.4.3	SHELL_BUFFER_SIZE	676
44.4.4	SHELL_MAX_ARGS	676
44.4.5	SHELL_HISTORY_COUNT	676
44.4.6	SHELL_HANDLE_SIZE	676
44.4.7	SHELL_COMMAND_DEFINE	676
44.4.8	SHELL_COMMAND	677
44.5	Typedef Documentation	677
44.5.1	cmd_function_t	677
44.6	Enumeration Type Documentation	677
44.6.1	shell_status_t	677
44.7	Function Documentation	677
44.7.1	SHELL_Init	677
44.7.2	SHELL_RegisterCommand	678
44.7.3	SHELL_UnregisterCommand	679
44.7.4	SHELL_Write	679
44.7.5	SHELL_Printf	679
44.7.6	SHELL_Task	680
 Chapter Fftx_cache_driver		
45.1	Overview	681
45.2	Data Structure Documentation	681
45.2.1	struct fftx_prefetch_speculation_status_t	681
45.2.2	struct fftx_cache_config_t	682
45.3	Macro Definition Documentation	682
45.3.1	FSL_FTFX_CACHE_DRIVER_VERSION	682
45.4	Enumeration Type Documentation	682
45.4.1	_ftfx_cache_ram_func_constants	682

Contents

Section Number	Title	Page Number
45.5	Function Documentation	682
45.5.1	FTFx_CACHE_Init	682
45.5.2	FTFx_CACHE_ClearCachePrefetchSpeculation	683
45.5.3	FTFx_CACHE_PflashSetPrefetchSpeculation	683
45.5.4	FTFx_CACHE_PflashGetPrefetchSpeculation	684
Chapter Ftfx_flash_driver		
46.1	Overview	685
46.2	Data Structure Documentation	689
46.2.1	union pflash_prot_status_t	689
46.2.2	struct flash_config_t	689
46.3	Macro Definition Documentation	689
46.3.1	FSL_FLASH_DRIVER_VERSION	689
46.4	Enumeration Type Documentation	689
46.4.1	_flash_driver_version_constants	689
46.4.2	flash_prot_state_t	690
46.4.3	flash_xacc_state_t	690
46.4.4	flash_property_tag_t	690
46.5	Function Documentation	691
46.5.1	FLASH_Init	691
46.5.2	FLASH_Erase	692
46.5.3	FLASH_EraseAll	693
46.5.4	FLASH_Program	694
46.5.5	FLASH_ProgramSection	695
46.5.6	FLASH_ReadResource	696
46.5.7	FLASH_VerifyErase	697
46.5.8	FLASH_VerifyEraseAll	698
46.5.9	FLASH_VerifyProgram	698
46.5.10	FLASH_GetSecurityState	699
46.5.11	FLASH_SecurityBypass	700
46.5.12	FLASH_SetFlexramFunction	700
46.5.13	FLASH_IsProtected	701
46.5.14	FLASH_IsExecuteOnly	702
46.5.15	FLASH_PflashSetProtection	703
46.5.16	FLASH_PflashGetProtection	703
46.5.17	FLASHGetProperty	704
Chapter Ftfx_flexnvm_driver		
47.1	Overview	705

Contents

Section Number	Title	Page Number
47.2	Data Structure Documentation	707
47.2.1	struct flexnvm_config_t	707
47.3	Macro Definition Documentation	708
47.3.1	FSL_FLEXNVM_DRIVER_VERSION	708
47.4	Enumeration Type Documentation	708
47.4.1	flexnvm_property_tag_t	708
47.5	Function Documentation	708
47.5.1	FLEXNVM_Init	708
47.5.2	FLEXNVM_DflashErase	709
47.5.3	FLEXNVM_EraseAll	710
47.5.4	FLEXNVM_DflashProgram	710
47.5.5	FLEXNVM_DflashProgramSection	711
47.5.6	FLEXNVM_ProgramPartition	712
47.5.7	FLEXNVM_ReadResource	713
47.5.8	FLEXNVM_DflashVerifyErase	714
47.5.9	FLEXNVM_VerifyEraseAll	715
47.5.10	FLEXNVM_DflashVerifyProgram	716
47.5.11	FLEXNVM_GetSecurityState	717
47.5.12	FLEXNVM_SecurityBypass	717
47.5.13	FLEXNVM_SetFlexramFunction	718
47.5.14	FLEXNVM_EepromWrite	719
47.5.15	FLEXNVM_DflashSetProtection	720
47.5.16	FLEXNVM_DflashGetProtection	720
47.5.17	FLEXNVM_EepromSetProtection	721
47.5.18	FLEXNVM_EepromGetProtection	721
47.5.19	FLEXNVMGetProperty	722

Chapter MMCCARD

48.1	Overview	723
48.2	Data Structure Documentation	724
48.2.1	struct mmccard_usr_param_t	724
48.2.2	struct mmc_card_t	724
48.3	Enumeration Type Documentation	726
48.3.1	_mmc_card_flag	726
48.4	Function Documentation	726
48.4.1	MMC_Init	726
48.4.2	MMC_Deinit	727
48.4.3	MMC_CardInit	727
48.4.4	MMC_CardDeinit	728

Contents

Section Number	Title	Page Number
48.4.5	MMC_HostInit	729
48.4.6	MMC_HostDeinit	729
48.4.7	MMC_HostReset	729
48.4.8	MMC_PowerOnCard	729
48.4.9	MMC_PowerOffCard	730
48.4.10	MMC_CheckReadOnly	730
48.4.11	MMC_ReadBlocks	730
48.4.12	MMC_WriteBlocks	731
48.4.13	MMC_EraseGroups	732
48.4.14	MMC_SelectPartition	732
48.4.15	MMC_SetBootConfig	733
48.4.16	MMC_StartBoot	733
48.4.17	MMC_SetBootConfigWP	734
48.4.18	MMC_ReadBootData	734
48.4.19	MMC_StopBoot	734
48.4.20	MMC_SetBootPartitionWP	735

Chapter SDCARD

49.1	Overview	737
49.2	Data Structure Documentation	738
49.2.1	struct sdcard_usr_param_t	738
49.2.2	struct sd_card_t	738
49.3	Enumeration Type Documentation	739
49.3.1	_sd_card_flag	739
49.4	Function Documentation	740
49.4.1	SD_Init	740
49.4.2	SD_Deinit	741
49.4.3	SD_CardInit	741
49.4.4	SD_CardDeinit	742
49.4.5	SD_HostInit	742
49.4.6	SD_HostDeinit	743
49.4.7	SD_HostReset	743
49.4.8	SD_PowerOnCard	743
49.4.9	SD_PowerOffCard	743
49.4.10	SD_WaitCardDetectStatus	744
49.4.11	SD_IsCardPresent	744
49.4.12	SD_CheckReadOnly	744
49.4.13	SD_SelectCard	745
49.4.14	SD_ReadStatus	745
49.4.15	SD_ReadBlocks	745
49.4.16	SD_WriteBlocks	746

Contents

Section Number		Page Number
	Title	
49.4.17	SD_EraseBlocks	747
49.4.18	SD_SetDriverStrength	748
49.4.19	SD_SetMaxCurrent	748

Chapter [SDIOCARD](#)

50.1	Overview	749
50.2	Data Structure Documentation	750
50.2.1	struct sdiocard_usr_param_t	750
50.2.2	struct sdio_card_t	750
50.3	Macro Definition Documentation	751
50.3.1	FSL_SDIO_DRIVER_VERSION	751
50.4	Function Documentation	751
50.4.1	SDIO_Init	751
50.4.2	SDIO_Deinit	752
50.4.3	SDIO_CardInit	752
50.4.4	SDIO_CardDeinit	753
50.4.5	SDIO_HostInit	754
50.4.6	SDIO_HostDeinit	754
50.4.7	SDIO_HostReset	754
50.4.8	SDIO_PowerOnCard	754
50.4.9	SDIO_PowerOffCard	755
50.4.10	SDIO_CardInActive	755
50.4.11	SDIO_IO_Write_Direct	755
50.4.12	SDIO_IO_Read_Direct	756
50.4.13	SDIO_IO_Write_Extended	756
50.4.14	SDIO_IO_Read_Extended	757
50.4.15	SDIO_GetCardCapability	758
50.4.16	SDIO_SetBlockSize	758
50.4.17	SDIO_CardReset	759
50.4.18	SDIO_SetDataBusWidth	759
50.4.19	SDIO_SwitchToHighSpeed	759
50.4.20	SDIO_ReadCIS	760
50.4.21	SDIO_EnableIOInterrupt	760
50.4.22	SDIO_EnableIO	761
50.4.23	SDIO_SelectIO	761
50.4.24	SDIO_AbortIO	762
50.4.25	SDIO_WaitCardDetectStatus	762
50.4.26	SDIO_IsCardPresent	762
50.4.27	SDIO_IO_Transfer	763

Contents

Section Number	Title	Page Number
Chapter	Data Structure Documentation	
51.0.28	ftfx_config_t Struct Reference	765
51.0.29	ftfx_ifr_desc_t Struct Reference	765
51.0.30	ftfx_mem_desc_t Struct Reference	765
51.0.31	ftfx_ops_config_t Struct Reference	766
51.0.32	ftfx_spec_mem_t Struct Reference	766
51.0.33	ftfx_swap_state_config_t Struct Reference	767
51.0.34	mmc_boot_config_t Struct Reference	767
51.0.35	mmc_cid_t Struct Reference	768
51.0.36	mmc_csd_t Struct Reference	769
51.0.37	mmc_extended_csd_config_t Struct Reference	770
51.0.38	mmc_extended_csd_t Struct Reference	771
51.0.39	sd_cid_t Struct Reference	775
51.0.40	sd_csd_t Struct Reference	776
51.0.41	sd_scr_t Struct Reference	777
51.0.42	sd_status_t Struct Reference	777
51.0.43	sdio_common_cis_t Struct Reference	778
51.0.44	sdio_fbr_t Struct Reference	779
51.0.45	sdio_func_cis_t Struct Reference	779

Chapter 1

Introduction

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

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

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

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

- IAR Embedded Workbench
- GNU Arm Embedded Toolchain

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

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

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

Table 2: MCUXpresso SDK 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_TxFrameOverLen` = 4003
- `kStatus_ENET_TxFrameBusy` = 4004
- `kStatus_ENET_TxFrameFail` = 4005
- `kStatus_ENET_PtpTsRingFull` = 4006
- `kStatus_ENET_PtpTsRingEmpty` = 4007
- `kStatus_FLEXCAN_TxBusy` = 5300
- `kStatus_FLEXCAN_TxIdle` = 5301
- `kStatus_FLEXCAN_TxSwitchToRx` = 5302
- `kStatus_FLEXCAN_RxBusy` = 5303
- `kStatus_FLEXCAN_RxIdle` = 5304
- `kStatus_FLEXCAN_RxOverflow` = 5305
- `kStatus_FLEXCAN_RxFifoBusy` = 5306
- `kStatus_FLEXCAN_RxFifoIdle` = 5307
- `kStatus_FLEXCAN_RxFifoOverflow` = 5308
- `kStatus_FLEXCAN_RxFifoWarning` = 5309
- `kStatus_FLEXCAN_ErrorStatus` = 5310
- `kStatus_FLEXCAN_UnHandled` = 5311
- `kStatus_I2C_Busy` = 1100
- `kStatus_I2C_Idle` = 1101
- `kStatus_I2C_Nak` = 1102
- `kStatus_I2C_ArbitrationLost` = 1103
- `kStatus_I2C_Timeout` = 1104
- `kStatus_I2C_Addr_Nak` = 1105
- `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_UART_TxBusy` = 1000

- `kStatus_UART_RxBusy` = 1001
- `kStatus_UART_TxIdle` = 1002
- `kStatus_UART_RxIdle` = 1003
- `kStatus_UART_TxWatermarkTooLarge` = 1004
- `kStatus_UART_RxWatermarkTooLarge` = 1005
- `kStatus_UART_FlagCannotClearManually` = 1006
- `kStatus_UART_Error` = 1007
- `kStatus_UART_RxRingBufferOverrun` = 1008
- `kStatus_UART_RxHardwareOverrun` = 1009
- `kStatus_UART_NoiseError` = 1010
- `kStatus_UART_FramingError` = 1011
- `kStatus_UART_ParityError` = 1012
- `kStatus_UART_BaudrateNotSupport` = 1013
- `kStatus_UART_IdleLineDetected` = 1014
- `kStatus_DMAMGR_ChannelOccupied` = 5200
- `kStatus_DMAMGR_ChannelNotUsed` = 5201
- `kStatus_DMAMGR_NoFreeChannel` = 5202
- `kStatus_NOTIFIER_ErrorNotificationBefore` = 9800
- `kStatus_NOTIFIER_ErrorNotificationAfter` = 9801

Chapter 3

Architectural Overview

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

Overview

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

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

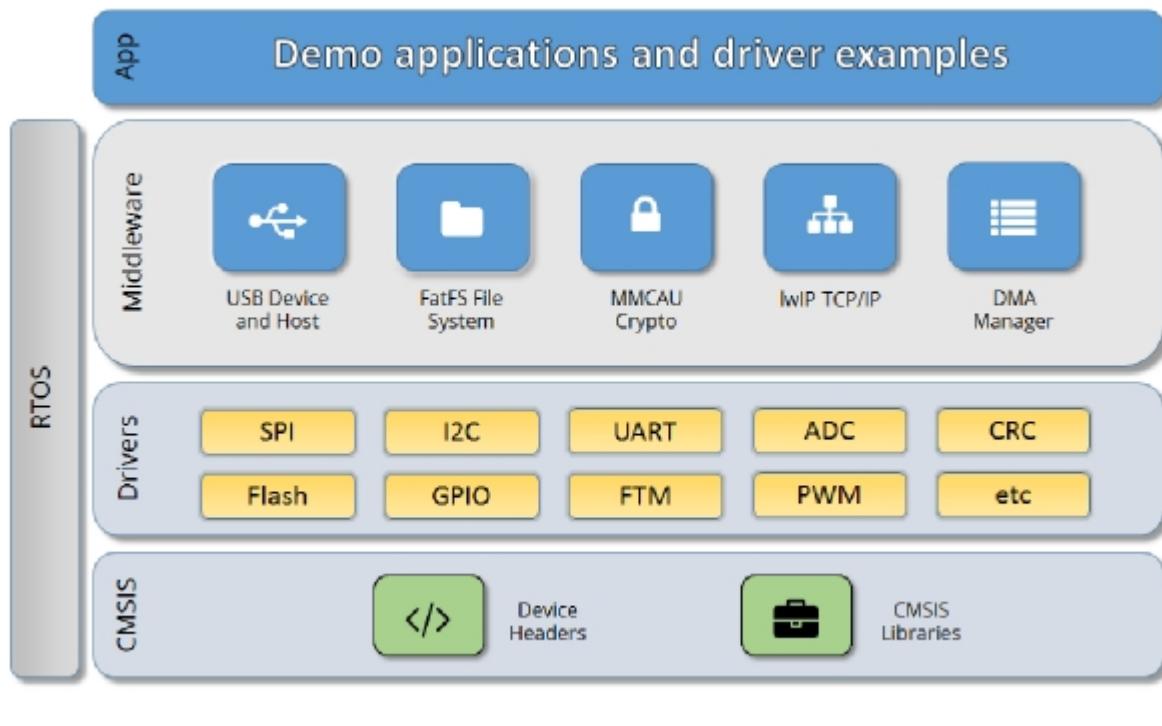


Figure 1: MCUXpresso SDK Block Diagram

MCU header files

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

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

CMSIS Support

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

MCUXpresso SDK Peripheral Drivers

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

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

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

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

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

Interrupt handling for transactional APIs

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

```
PUBWEAK SPI0_IRQHandler  
PUBWEAK SPI0_DriverIRQHandler  
SPI0_IRQHandler
```

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

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

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

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

Feature Header Files

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

Application

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



Chapter 4 Trademarks

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

How to Reach Us:

Home Page: [nxp.com](http://www.nxp.com)

Web Support: [nxp.com/support](http://www.nxp.com/support)

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

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

© 2018 NXP B.V.

Chapter 5

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

5.1 Overview

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

5.2 Typical use case

5.2.1 Polling Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/adc16

5.2.2 Interrupt Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/adc16

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_COCAO_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.

Typical use case

- 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 }
 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)

Data Structure Documentation

Gets the status flags of channel.

5.3 Data Structure Documentation

5.3.1 struct adc16_config_t

Data Fields

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

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 `adc16_reference_voltage_source_t adc16_config_t::referenceVoltageSource`

5.3.1.0.0.1.2 `adc16_clock_source_t adc16_config_t::clockSource`

5.3.1.0.0.1.3 `bool adc16_config_t::enableAsynchronousClock`

5.3.1.0.0.1.4 `adc16_clock_divider_t adc16_config_t::clockDivider`

5.3.1.0.0.1.5 `adc16_resolution_t adc16_config_t::resolution`

5.3.1.0.0.1.6 `adc16_long_sample_mode_t adc16_config_t::longSampleMode`

5.3.1.0.0.1.7 `bool adc16_config_t::enableHighSpeed`

5.3.1.0.0.1.8 `bool adc16_config_t::enableLowPower`

5.3.1.0.0.1.9 `bool adc16_config_t::enableContinuousConversion`

5.3.2 struct `adc16_hardware_compare_config_t`

Data Fields

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

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 `adc16_hardware_compare_mode_t adc16_hardware_compare_config_t::hardwareCompareMode`

See "adc16_hardware_compare_mode_t".

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`

Enumeration Type Documentation

- **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.

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.

Function Documentation

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)

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.

Function Documentation

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]

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.

Function Documentation

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

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.

Function Documentation

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".

Chapter 6

CMP: Analog Comparator Driver

6.1 Overview

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

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

6.2 Typical use case

6.2.1 Polling Configuration

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

6.2.2 Interrupt Configuration

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

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.

Typical use case

- 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.
- 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.

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

6.3.2 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.

Enumeration Type Documentation

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.

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.

Function Documentation

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

Parameters

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

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

Parameters

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

6.6.4 void CMP_GetDefaultConfig (**cmp_config_t** * *config*)

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

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

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.
-------------	------------------------------

Function Documentation

<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*)

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 MCUXpresso SDK provides a driver for the CMT module of the MCUXpresso SDK 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.

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/cmt
This is an example IRQ handler to change the mark and space count to complete data modulation.

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

Data Structures

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

Typical use case

- 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)
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.

Data Structure Documentation

- `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.

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.

Function Documentation

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.

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>modulate-Config</i>	The carrier generation and modulator configuration.

7.7.5 **cmt_mode_t** CMT_GetMode (**CMT_Type** * *base*)

Function Documentation

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*)**

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 (*markCount* + 1) / (Fcmt/8). The space period of the generated signal equals *spaceCount* / (Fcmt/8).
- In FSK mode: The mark period of the generated signal equals (*markCount* + 1)/fcg. The space period of the generated signal equals *spaceCount* / 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.

Function Documentation

**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.

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);
*
```

Function Documentation

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:

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

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

8.2 CRC Driver Initialization and Configuration

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

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

8.3 CRC Write Data

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

8.4 CRC Get Checksum

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

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

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

Comments about API usage in RTOS

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

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

8.5 Comments about API usage in RTOS

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

The triplets

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

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

Data Structures

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

Macros

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

Enumerations

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

Functions

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

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

Driver version

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

8.6 Data Structure Documentation

8.6.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

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

8.6.1.0.0.9 Field Documentation

8.6.1.0.0.9.1 uint32_t crc_config_t::polynomial

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

8.6.1.0.0.9.2 bool crc_config_t::reflectIn

8.6.1.0.0.9.3 bool crc_config_t::reflectOut

8.6.1.0.0.9.4 bool crc_config_t::complementChecksum

8.6.1.0.0.9.5 crc_bits_t crc_config_t::crcBits

Function Documentation

8.7 Macro Definition Documentation

8.7.1 `#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

8.7.2 `#define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1`

Use CRC16-CCIT-FALSE as default.

8.8 Enumeration Type Documentation

8.8.1 `enum crc_bits_t`

Enumerator

`kCrcBits16` Generate 16-bit CRC code.

`kCrcBits32` Generate 32-bit CRC code.

8.8.2 `enum crc_result_t`

Enumerator

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

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

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

8.9 Function Documentation

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

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

Parameters

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

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

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

Parameters

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

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

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

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

Parameters

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

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

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

Parameters

Function Documentation

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

8.9.5 `uint32_t CRC_Get32bitResult (CRC_Type * base)`

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

Parameters

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

Returns

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

8.9.6 `uint16_t CRC_Get16bitResult (CRC_Type * base)`

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

Parameters

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

Returns

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

Chapter 9

DAC: Digital-to-Analog Converter Driver

9.1 Overview

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

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

9.2.2 Working with the hardware buffer

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

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 }

Typical use case

- 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.

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.

Enumeration Type Documentation

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.

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.

Function Documentation

- 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.
<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*)

Function Documentation

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.

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*)

Function Documentation

Parameters

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

Chapter 10

DMAMUX: Direct Memory Access Multiplexer Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 DMAMUX Operation

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

Driver version

- #define **FSL_DMAMUX_DRIVER_VERSION** (MAKE_VERSION(2, 0, 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 MCUXpresso SDK provides a peripheral driver for the Serial Peripheral Interface (SPI) module of MCUXpresso SDK devices.

Modules

- DSPI DMA Driver
- DSPI Driver
- DSPI FreeRTOS Driver
- DSPI eDMA Driver

DSPI Driver

11.2 DSPI Driver

11.2.1 Overview

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

11.2.2 Typical use case

11.2.2.1 Master Operation

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

11.2.2.2 Slave Operation

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

Data Structures

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

Macros

- #define [DSPI_DUMMY_DATA](#) (0x00U)
DSPI dummy data if there is no Tx data.
- #define [DSPI_MASTER_CTAR_SHIFT](#) (0U)
DSPI master CTAR shift macro; used internally.

- #define **DSPI_MASTER_CTAR_MASK** (0x0FU)
DSPI master CTAR mask macro; used internally.
- #define **DSPI_MASTER_PCS_SHIFT** (4U)
DSPI master PCS shift macro; used internally.
- #define **DSPI_MASTER_PCS_MASK** (0xF0U)
DSPI master PCS mask macro; used internally.
- #define **DSPI_SLAVE_CTAR_SHIFT** (0U)
DSPI slave CTAR shift macro; used internally.
- #define **DSPI_SLAVE_CTAR_MASK** (0x07U)
DSPI slave CTAR mask macro; used internally.

Typedefs

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

Enumerations

- enum **_dspi_status** {

kStatus_DSPI_Busy = MAKE_STATUS(kStatusGroup_DSPI, 0),

kStatus_DSPI_Error = MAKE_STATUS(kStatusGroup_DSPI, 1),

kStatus_DSPI_Idle = MAKE_STATUS(kStatusGroup_DSPI, 2),

kStatus_DSPI_OutOfRange = MAKE_STATUS(kStatusGroup_DSPI, 3) }

Status for the DSPI driver.
- enum **_dspi_flags** {

kDSPI_TxCompleteFlag = (int)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 = (int)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,

DSPI Driver

- ```
kDSPI_AllInterruptEnable }

DSPI interrupt source.
• enum _dspi_dma_enable {
 kDSPI_TxDmaEnable = (SPI_RSER_TFFF_RE_MASK | SPI_RSER_TFFF_DIRS_MASK),
 kDSPI_RxDmaEnable = (SPI_RSER_RFDF_RE_MASK | SPI_RSER_RFDF_DIRS_MASK) }

DSPI DMA source.
• enum dspi_master_slave_mode_t {
 kDSPI_Master = 1U,
 kDSPI_Slave = 0U }

DSPI master or slave mode configuration.
• enum dspi_master_sample_point_t {
 kDSPI_SckToSin0Clock = 0U,
 kDSPI_SckToSin1Clock = 1U,
 kDSPI_SckToSin2Clock = 2U }

DSPI Sample Point: Controls when the DSPI master samples SIN in the Modified Transfer Format.
• enum dspi_which_pcs_t {
 kDSPI_Pcs0 = 1U << 0,
 kDSPI_Pcs1 = 1U << 1,
 kDSPI_Pcs2 = 1U << 2,
 kDSPI_Pcs3 = 1U << 3,
 kDSPI_Pcs4 = 1U << 4,
 kDSPI_Pcs5 = 1U << 5 }

DSPI Peripheral Chip Select (Pcs) configuration (which Pcs to configure).
• enum dspi_pcs_polarity_config_t {
 kDSPI_PcsActiveHigh = 0U,
 kDSPI_PcsActiveLow = 1U }

DSPI Peripheral Chip Select (Pcs) Polarity configuration.
• enum _dspi_pcs_polarity {
 kDSPI_Pcs0ActiveLow = 1U << 0,
 kDSPI_Pcs1ActiveLow = 1U << 1,
 kDSPI_Pcs2ActiveLow = 1U << 2,
 kDSPI_Pcs3ActiveLow = 1U << 3,
 kDSPI_Pcs4ActiveLow = 1U << 4,
 kDSPI_Pcs5ActiveLow = 1U << 5,
 kDSPI_PcsAllActiveLow = 0xFFU }

DSPI Peripheral Chip Select (Pcs) Polarity.
• enum dspi_clock_polarity_t {
 kDSPI_ClockPolarityActiveHigh = 0U,
 kDSPI_ClockPolarityActiveLow = 1U }

DSPI clock polarity configuration for a given CTAR.
• enum dspi_clock_phase_t {
 kDSPI_ClockPhaseFirstEdge = 0U,
 kDSPI_ClockPhaseSecondEdge = 1U }

DSPI clock phase configuration for a given CTAR.
• enum dspi_shift_direction_t {
 kDSPI_MsbFirst = 0U,
 kDSPI_LsbFirst = 1U }
```

*DSPI data shifter direction options for a given CTAR.*

- enum `dspi_delay_type_t` {
   
    `kDSPI_PesToSck` = 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` }

*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.*

## DSPI Driver

### Variables

- volatile uint8\_t `g_dspiDummyData []`  
*Global variable for dummy data value setting.*

### Driver version

- #define `FSL_DSPI_DRIVER_VERSION` (MAKE\_VERSION(2, 2, 0))  
*DSPI driver version 2.2.0.*

### 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.*
- 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

- uint32\_t **DSPIGetInstance** (SPI\_Type \*base)  
*Get instance number for DSPI module.*
- 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.*

## DSPI Driver

- 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.*
- void **DSPI\_SetDummyData** (SPI\_Type \*base, uint8\_t dummyData)  
*Set up the dummy data.*

## 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\_MasterHalfDuplexTransferBlocking** (SPI\_Type \*base, **dspi\_half\_duplex\_transfer\_t** \*xfer)  
*Transfers a block of data using a polling method.*
- status\_t **DSPI\_MasterHalfDuplexTransferNonBlocking** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle, **dspi\_half\_duplex\_transfer\_t** \*xfer)  
*Performs a non-blocking DSPI interrupt transfer.*
- 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](#)  
*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.*

## DSPI Driver

### 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.*

#### 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_fifoOverWrite`**

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.

## DSPI Driver

### 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*

*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\_half\_duplex\_transfer\_t

##### Data Fields

- **uint8\_t \* txData**  
*Send buffer.*
- **uint8\_t \* rxData**  
*Receive buffer.*
- **size\_t txDataSize**  
*Transfer bytes for transmit.*
- **size\_t rxDataSize**  
*Transfer bytes.*
- **uint32\_t configFlags**  
*Transfer configuration flags; set from \_dspi\_transfer\_config\_flag\_for\_master.*
- **bool isPcsAssertInTransfer**  
*If Pcs pin keep assert between transmit and receive.*
- **bool isTransmitFirst**  
*True for transmit first and false for receive first.*

#### 11.2.3.7.0.18 Field Documentation

**11.2.3.7.0.18.1 uint32\_t dspi\_half\_duplex\_transfer\_t::configFlags**

**11.2.3.7.0.18.2 bool dspi\_half\_duplex\_transfer\_t::isPcsAssertInTransfer**

true for assert and false for deassert.

**11.2.3.7.0.18.3 bool dspi\_half\_duplex\_transfer\_t::isTransmitFirst**

#### 11.2.3.8 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**

## DSPI Driver

- **volatile uint32\_t lastCommand**  
*The desired 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.*

### 11.2.3.8.0.19 Field Documentation

- 11.2.3.8.0.19.1 `uint32_t dspi_master_handle_t::bitsPerFrame`
- 11.2.3.8.0.19.2 `volatile uint32_t dspi_master_handle_t::command`
- 11.2.3.8.0.19.3 `volatile uint32_t dspi_master_handle_t::lastCommand`
- 11.2.3.8.0.19.4 `uint8_t dspi_master_handle_t::fifoSize`
- 11.2.3.8.0.19.5 `volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer`
- 11.2.3.8.0.19.6 `volatile bool dspi_master_handle_t::isThereExtraByte`
- 11.2.3.8.0.19.7 `uint8_t* volatile dspi_master_handle_t::txData`
- 11.2.3.8.0.19.8 `uint8_t* volatile dspi_master_handle_t::rxData`
- 11.2.3.8.0.19.9 `volatile size_t dspi_master_handle_t::remainingSendByteCount`
- 11.2.3.8.0.19.10 `volatile size_t dspi_master_handle_t::remainingReceiveByteCount`
- 11.2.3.8.0.19.11 `volatile uint8_t dspi_master_handle_t::state`
- 11.2.3.8.0.19.12 `dspi_master_transfer_callback_t dspi_master_handle_t::callback`
- 11.2.3.8.0.19.13 `void* dspi_master_handle_t::userData`

### 11.2.3.9 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.*

### 11.2.3.9.0.20 Field Documentation

11.2.3.9.0.20.1 `uint32_t dspi_slave_handle_t::bitsPerFrame`

11.2.3.9.0.20.2 `volatile bool dspi_slave_handle_t::isThereExtraByte`

11.2.3.9.0.20.3 `uint8_t* volatile dspi_slave_handle_t::txData`

11.2.3.9.0.20.4 `uint8_t* volatile dspi_slave_handle_t::rxData`

11.2.3.9.0.20.5 `volatile size_t dspi_slave_handle_t::remainingSendByteCount`

11.2.3.9.0.20.6 `volatile size_t dspi_slave_handle_t::remainingReceiveByteCount`

11.2.3.9.0.20.7 `volatile uint8_t dspi_slave_handle_t::state`

11.2.3.9.0.20.8 `volatile uint32_t dspi_slave_handle_t::errorCount`

11.2.3.9.0.20.9 `dspi_slave_transfer_callback_t dspi_slave_handle_t::callback`

11.2.3.9.0.20.10 `void* dspi_slave_handle_t::userData`

### 11.2.4 Macro Definition Documentation

11.2.4.1 `#define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))`

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 dsPIC33EP DSPI API Reference</a> . |
| <i>srcClock_Hz</i>  | Module source input clock in Hertz.                                              |

#### 11.2.7.2 void DSPI\_MasterGetDefaultConfig ( [dsPIC33F dsPIC33EP 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 dsPIC33EP DSPI API Reference
* DSPI_MasterGetDefaultConfig(&masterConfig);
*

```

#### Parameters

|                     |                                                                            |
|---------------------|----------------------------------------------------------------------------|
| <i>masterConfig</i> | pointer to <a href="#">dsPIC33F dsPIC33EP DSPI API Reference</a> structure |
|---------------------|----------------------------------------------------------------------------|

#### 11.2.7.3 void DSPI\_SlaveInit ( [SPI\\_Type](#) \* *base*, const [dsPIC33F dsPIC33EP DSPI API Reference](#) \* *slaveConfig* )

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

```

* dsPIC33F dsPIC33EP DSPI API Reference
* 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. Do not use this API(write to RSER register) while DSPI is in running state.

```
* 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 **uint32\_t DSPI\_GetInstance ( SPI\_Type \* *base* )**

Parameters

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

### 11.2.7.17 **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. |

### 11.2.7.18 **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.

### 11.2.7.19 **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. |
|-------------|--------------------------|

#### 11.2.7.20 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. |
|-------------|--------------------------|

#### 11.2.7.21 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 |

#### 11.2.7.22 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 |

#### 11.2.7.23 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 Driver

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

Parameters

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

### 11.2.7.24 `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 dspi_ctar_selection_t |
| <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

### 11.2.7.25 `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>                         |

**11.2.7.26 `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>delayTimeIn-NanoSec</i> | The desired delay value in nanoseconds.                                                                |

## DSPI Driver

Returns

The actual calculated delay value.

### 11.2.7.27 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 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 = kDSPI_Ctar0;
* commandConfig.whichPcs = kDSPI_Pcs0;
* 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.         |

### 11.2.7.28 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 <code>dspi_command_data_config_t</code> structure. |
|----------------|-------------------------------------------------------------------|

### 11.2.7.29 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.         |

### 11.2.7.30 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.

## DSPI Driver

Parameters

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

Returns

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

### 11.2.7.31 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 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. |

**11.2.7.32 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.        |

**11.2.7.33 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.        |

**11.2.7.34 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.

**11.2.7.35 void DSPI\_SetDummyData ( SPI\_Type \* *base*, uint8\_t *dummyData* )**

## DSPI Driver

Parameters

|                  |                                                |
|------------------|------------------------------------------------|
| <i>base</i>      | DSPI peripheral address.                       |
| <i>dummyData</i> | Data to be transferred when tx buffer is NULL. |

**11.2.7.36 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.                 |

**11.2.7.37 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.

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.

**11.2.7.38 status\_t DSPI\_MasterTransferNonBlocking ( SPI\_Type \* *base*, dspi\_master\_handle\_t \* *handle*, dspi\_transfer\_t \* *transfer* )**

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

Parameters

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

Returns

status of `status_t`.

#### 11.2.7.39 `status_t DSPI_MasterHalfDuplexTransferBlocking ( SPI_Type * base, dspi_half_duplex_transfer_t * xfer )`

This function will do a half-duplex transfer for DSPI master, This is a blocking function, which does not return until all transfer have been completed. And data transfer will be half-duplex, users can set transmit first or receive first.

Parameters

|             |                                                               |
|-------------|---------------------------------------------------------------|
| <i>base</i> | DSPI base pointer                                             |
| <i>xfer</i> | pointer to <code>dspi_half_duplex_transfer_t</code> structure |

Returns

status of `status_t`.

#### 11.2.7.40 `status_t DSPI_MasterHalfDuplexTransferNonBlocking ( SPI_Type * base, dspi_master_handle_t * handle, dspi_half_duplex_transfer_t * xfer )`

This function transfers data using interrupts, the transfer mechanism is half-duplex. 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 <code>dspi_master_handle_t</code> structure which stores the transfer state |

## DSPI Driver

|             |                                                               |
|-------------|---------------------------------------------------------------|
| <i>xfer</i> | pointer to <code>dspi_half_duplex_transfer_t</code> structure |
|-------------|---------------------------------------------------------------|

Returns

status of `status_t`.

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

This function gets the master transfer count.

Parameters

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

Returns

status of `status_t`.

### 11.2.7.42 `void DSPI_MasterTransferAbort ( SPI_Type * base, dspi_master_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_master_handle_t</code> structure which stores the transfer state. |

### 11.2.7.43 `void DSPI_MasterTransferHandleIRQ ( SPI_Type * base, dspi_master_handle_t * handle )`

This function processes the DSPI transmit and receive IRQ.

Parameters

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

#### 11.2.7.44 `void DSPI_SlaveTransferCreateHandle ( SPI_Type * base, dspi_slave_handle_t * handle, dspi_slave_transfer_callback_t callback, void * userData )`

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

Parameters

|                 |                                                               |
|-----------------|---------------------------------------------------------------|
| <i>handle</i>   | DSPI handle pointer to the <code>dspi_slave_handle_t</code> . |
| <i>base</i>     | DSPI peripheral base address.                                 |
| <i>callback</i> | DSPI callback.                                                |
| <i>userData</i> | Callback function parameter.                                  |

#### 11.2.7.45 `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`.

#### 11.2.7.46 `status_t DSPI_SlaveTransferGetCount ( SPI_Type * base, dspi_slave_handle_t * handle, size_t * count )`

This function gets the slave transfer count.

## 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. |
| <i>count</i>  | The number of bytes transferred by using the non-blocking transaction.         |

Returns

status of status\_t.

### 11.2.7.47 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 dspi_slave_handle_t structure which stores the transfer state. |

### 11.2.7.48 void DSPI\_SlaveTransferHandleIRQ ( SPI\_Type \* *base*, dspi\_slave\_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_slave_handle_t structure which stores the transfer state. |

## 11.2.8 Variable Documentation

### 11.2.8.1 volatile uint8\_t g\_dspiDummyData[]

## 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.*

## Driver version

- #define **FSL\_DSPI\_DMA\_DRIVER\_VERSION** (MAKE\_VERSION(2, 2, 0))  
*DSPI DMA driver version 2.2.0.*

### 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*

### 11.3.2.1.0.21 Field Documentation

**11.3.2.1.0.21.1 `uint32_t dspi_master_dma_handle_t::bitsPerFrame`**

**11.3.2.1.0.21.2 `volatile uint32_t dspi_master_dma_handle_t::command`**

**11.3.2.1.0.21.3 `volatile uint32_t dspi_master_dma_handle_t::lastCommand`**

**11.3.2.1.0.21.4 `uint8_t dspi_master_dma_handle_t::fifoSize`**

**11.3.2.1.0.21.5 `volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer`**

**11.3.2.1.0.21.6 `volatile bool dspi_master_dma_handle_t::isThereExtraByte`**

**11.3.2.1.0.21.7 `uint8_t* volatile dspi_master_dma_handle_t::txData`**

**11.3.2.1.0.21.8 `uint8_t* volatile dspi_master_dma_handle_t::rxData`**

**11.3.2.1.0.21.9 `volatile size_t dspi_master_dma_handle_t::remainingSendByteCount`**

**11.3.2.1.0.21.10 `volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount`**

**11.3.2.1.0.21.11 `uint32_t dspi_master_dma_handle_t::rxBuffIfNull`**

**11.3.2.1.0.21.12 `uint32_t dspi_master_dma_handle_t::txBuffIfNull`**

**11.3.2.1.0.21.13 `volatile uint8_t dspi_master_dma_handle_t::state`**

**11.3.2.1.0.21.14 `dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback`**

**11.3.2.1.0.21.15 `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`

## DSPI DMA Driver

- **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**
  - 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*

### 11.3.2.2.0.22 Field Documentation

- 11.3.2.2.0.22.1 `uint32_t dspi_slave_dma_handle_t::bitsPerFrame`
- 11.3.2.2.0.22.2 `volatile bool dspi_slave_dma_handle_t::isThereExtraByte`
- 11.3.2.2.0.22.3 `uint8_t* volatile dspi_slave_dma_handle_t::txData`
- 11.3.2.2.0.22.4 `uint8_t* volatile dspi_slave_dma_handle_t::rxData`
- 11.3.2.2.0.22.5 `volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount`
- 11.3.2.2.0.22.6 `volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount`
- 11.3.2.2.0.22.7 `uint32_t dspi_slave_dma_handle_t::rxBuffIfNull`
- 11.3.2.2.0.22.8 `uint32_t dspi_slave_dma_handle_t::txBuffIfNull`
- 11.3.2.2.0.22.9 `uint32_t dspi_slave_dma_handle_t::txLastData`
- 11.3.2.2.0.22.10 `volatile uint8_t dspi_slave_dma_handle_t::state`
- 11.3.2.2.0.22.11 `uint32_t dspi_slave_dma_handle_t::errorCount`
- 11.3.2.2.0.22.12 `dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback`
- 11.3.2.2.0.22.13 `void* dspi_slave_dma_handle_t::userData`

### 11.3.3 Macro Definition Documentation

- 11.3.3.1 `#define FSL_DSPI_DMA_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))`

### 11.3.4 Typedef Documentation

- 11.3.4.1 `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.   |

### 11.3.4.2 **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.   |

## 11.3.5 Function Documentation

### 11.3.5.1 **void DSPI\_MasterTransferCreateHandleDMA ( SPI\_Type \* *base*, dspi\_master\_dma\_handle\_t \* *handle*, dspi\_master\_dma\_transfer\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *dmaRxRegToRxDataHandle*, dma\_handle\_t \* *dmaTxDataToIntermediaryHandle*, dma\_handle\_t \* *dmaIntermediaryToTxRegHandle* )**

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

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for *dmaRxRegToRxDataHandle* and Tx DMAMUX source for *dmaIntermediaryToTxRegHandle*. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for *dmaRxRegToRxDataHandle*.

Parameters

|                                        |                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------|
| <i>base</i>                            | DSPI peripheral base address.                                                     |
| <i>handle</i>                          | DSPI handle pointer to <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> .  |

#### 11.3.5.2 `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`.

#### 11.3.5.3 `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. |

### 11.3.5.4 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.

### 11.3.5.5 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. |

#### 11.3.5.6 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.

#### 11.3.5.7 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. |
|---------------|-------------------------------------------------------------------------------------|

### 11.3.5.8 **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.*
- status\_t [DSPI\\_MasterHalfDuplexTransferEDMA](#) (SPI\_Type \*base, dspi\_master\_edma\_handle\_t \*handle, [dspi\\_half\\_duplex\\_transfer\\_t](#) \*xfer)  
*Transfers a block of data using a eDMA method.*
- 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.*

## DSPI eDMA Driver

- status\_t **DSPI\_SlaveTransferEDMA** (SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle, **dspi\_transfer\_t** \*transfer)  
*DSPI slave transfer data using eDMA.*
- 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.*

## Driver version

- #define **FSL\_DSPI\_EDMA\_DRIVER\_VERSION** (MAKE\_VERSION(2, 2, 0))  
*DSPI EDMA driver version 2.2.0.*

### 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**

- `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 dspiSoftwareTCD [2]`  
*SoftwareTCD , internal used.*

#### 11.4.2.1.0.23 Field Documentation

11.4.2.1.0.23.1 `uint32_t dspi_master_edma_handle_t::bitsPerFrame`

11.4.2.1.0.23.2 `volatile uint32_t dspi_master_edma_handle_t::command`

11.4.2.1.0.23.3 `volatile uint32_t dspi_master_edma_handle_t::lastCommand`

11.4.2.1.0.23.4 `uint8_t dspi_master_edma_handle_t::fifoSize`

11.4.2.1.0.23.5 `volatile bool dspi_master_edma_handle_t::isPcsActiveAfterTransfer`

11.4.2.1.0.23.6 `uint8_t dspi_master_edma_handle_t::nbytes`

11.4.2.1.0.23.7 `volatile uint8_t dspi_master_edma_handle_t::state`

11.4.2.1.0.23.8 `uint8_t* volatile dspi_master_edma_handle_t::txData`

11.4.2.1.0.23.9 `uint8_t* volatile dspi_master_edma_handle_t::rxData`

11.4.2.1.0.23.10 `volatile size_t dspi_master_edma_handle_t::remainingSendByteCount`

11.4.2.1.0.23.11 `volatile size_t dspi_master_edma_handle_t::remainingReceiveByteCount`

11.4.2.1.0.23.12 `uint32_t dspi_master_edma_handle_t::rxBuffIfNull`

11.4.2.1.0.23.13 `uint32_t dspi_master_edma_handle_t::txBuffIfNull`

11.4.2.1.0.23.14 `dspi_master_edma_transfer_callback_t dspi_master_edma_handle_t::callback`

11.4.2.1.0.23.15 `void* dspi_master_edma_handle_t::userData`

#### 11.4.2.2 `struct _dspi_slave_edma_handle`

Forward declaration of the DSPI eDMA slave handle typedefs.

### Data Fields

- `uint32_t bitsPerFrame`  
*The desired number of bits per frame.*
- `uint8_t *volatile txData`  
*Send buffer.*
- `uint8_t *volatile rxData`  
*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*

#### 11.4.2.2.0.24 Field Documentation

- 11.4.2.2.0.24.1 `uint32_t dspi_slave_edma_handle_t::bitsPerFrame`
- 11.4.2.2.0.24.2 `uint8_t* volatile dspi_slave_edma_handle_t::txData`
- 11.4.2.2.0.24.3 `uint8_t* volatile dspi_slave_edma_handle_t::rxData`
- 11.4.2.2.0.24.4 `volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount`
- 11.4.2.2.0.24.5 `volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount`
- 11.4.2.2.0.24.6 `uint32_t dspi_slave_edma_handle_t::rxBuffIfNull`
- 11.4.2.2.0.24.7 `uint32_t dspi_slave_edma_handle_t::txBuffIfNull`
- 11.4.2.2.0.24.8 `uint32_t dspi_slave_edma_handle_t::txLastData`
- 11.4.2.2.0.24.9 `uint8_t dspi_slave_edma_handle_t::nbytes`
- 11.4.2.2.0.24.10 `volatile uint8_t dspi_slave_edma_handle_t::state`
- 11.4.2.2.0.24.11 `dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback`
- 11.4.2.2.0.24.12 `void* dspi_slave_edma_handle_t::userData`

#### 11.4.3 Macro Definition Documentation

- 11.4.3.1 `#define FSL_DSPI_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))`

#### 11.4.4 Typedef Documentation

- 11.4.4.1 `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. |

### 11.4.4.2 **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. |

## 11.4.5 Function Documentation

### 11.4.5.1 **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> .  |

#### 11.4.5.2 `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`.

#### 11.4.5.3 `status_t DSPI_MasterHalfDuplexTransferEDMA ( SPI_Type * base, dspi_master_edma_handle_t * handle, dspi_half_duplex_transfer_t * xfer )`

This function transfers data using eDNA, the transfer mechanism is half-duplex. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

## DSPI eDMA Driver

Parameters

|                 |                                                                                                    |
|-----------------|----------------------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI base pointer                                                                                  |
| <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_half_duplex_transfer_t</code> structure.                               |

Returns

status of `status_t`.

### 11.4.5.4 void DSPI\_MasterTransferAbortEDMA ( `SPI_Type * base,` `dspi_master_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_master_edma_handle_t</code> structure which stores the transfer state. |

### 11.4.5.5 `status_t DSPI_MasterTransferGetCountEDMA ( SPI_Type * base,` `dspi_master_edma_handle_t * handle, size_t * count )`

This function gets the master eDMA transfer count.

Parameters

|               |                                                                                                    |
|---------------|----------------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                                      |
| <i>handle</i> | A pointer to the <code>dspi_master_edma_handle_t</code> structure which stores the transfer state. |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction.                                     |

Returns

status of `status_t`.

### 11.4.5.6 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 <a href="#">dspi_slave_edma_handle_t</a> .  |
| <i>callback</i>                | DSPI callback.                                                     |
| <i>userData</i>                | A callback function parameter.                                     |
| <i>edmaRxRegToRxDataHandle</i> | edmaRxRegToRxDataHandle pointer to <a href="#">edma_handle_t</a> . |
| <i>edmaTxDataToTxRegHandle</i> | edmaTxDataToTxRegHandle pointer to <a href="#">edma_handle_t</a> . |

#### **11.4.5.7 status\_t DSPI\_SlaveTransferEDMA ( SPI\_Type \* *base*, dsPIC\_Slave\_Edma\_Handle\_t \* *handle*, dsPIC\_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 <a href="#">dsPIC_Slave_Edma_Handle_t</a> structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <a href="#">dsPIC_Transfer_t</a> structure.                                          |

Returns

status of [status\\_t](#).

#### **11.4.5.8 void DSPI\_SlaveTransferAbortEDMA ( SPI\_Type \* *base*, dsPIC\_Slave\_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_slave_edma_handle_t structure which stores the transfer state. |

### 11.4.5.9 **status\_t DSPI\_SlaveTransferGetCountEDMA ( SPI\_Type \* *base*, dspi\_slave\_edma\_handle\_t \* *handle*, size\_t \* *count* )**

This function gets the slave eDMA transfer count.

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

#### Driver version

- #define `FSL_DSPI_FREERTOS_DRIVER_VERSION` (MAKE\_VERSION(2, 2, 0))  
*DSPI FREERTOS driver version 2.2.0.*

#### 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 Macro Definition Documentation

#### 11.5.2.1 #define FSL\_DSPI\_FREERTOS\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 0))

### 11.5.3 Function Documentation

#### 11.5.3.1 status\_t DSPI\_RTOS\_Init ( `dspi_rtos_handle_t * handle`, `SPI_Type * base`, `const dsPICMasterConfig_t * masterConfig`, `uint32_t srcClock_Hz` )

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

Parameters

|                           |                                                                           |
|---------------------------|---------------------------------------------------------------------------|
| <code>handle</code>       | The RTOS DSPI handle, the pointer to an allocated space for RTOS context. |
| <code>base</code>         | The pointer base address of the DSPI instance to initialize.              |
| <code>masterConfig</code> | A configuration structure to set-up the DSPI in master mode.              |
| <code>srcClock_Hz</code>  | A frequency of the input clock of the DSPI module.                        |

Returns

status of the operation.

### 11.5.3.2 `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. |
|---------------|-----------------------|

### 11.5.3.3 **status\_t DSPI\_RTOS\_Transfer( dsPIC33F\_RTOS\_Handle \* handle, dsPIC\_Transfer\_t \* transfer )**

This function performs the SPI transfer according to the data given in the transfer structure.

Parameters

|                 |                                                 |
|-----------------|-------------------------------------------------|
| <i>handle</i>   | The RTOS DSPI handle.                           |
| <i>transfer</i> | A structure specifying the transfer parameters. |

Returns

status of the operation.



# Chapter 12

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

### 12.1 Overview

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

### 12.2 Typical use case

#### 12.2.1 eDMA Operation

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

### Data Structures

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

### Macros

- #define `DMA_DCHPRI_INDEX`(channel) (((channel) & ~0x03U) | (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.*

### Typedefs

- typedef void(\* `edma_callback` )(struct \_edma\_handle \*handle, void \*userData, bool transferDone, uint32\_t tcds)  
*Define callback function for eDMA.*

## Typical use case

### Enumerations

- enum `edma_transfer_size_t` {  
    `kEDMA_TransferSize1Bytes` = 0x0U,  
    `kEDMA_TransferSize2Bytes` = 0x1U,  
    `kEDMA_TransferSize4Bytes` = 0x2U,  
    `kEDMA_TransferSize8Bytes` = 0x3U,  
    `kEDMA_TransferSize16Bytes` = 0x4U,  
    `kEDMA_TransferSize32Bytes` = 0x5U }

*eDMA transfer configuration*

- enum `edma_modulo_t` {  
    `kEDMA_ModuloDisable` = 0x0U,  
    `kEDMA_Modulo2bytes`,  
    `kEDMA_Modulo4bytes`,  
    `kEDMA_Modulo8bytes`,  
    `kEDMA_Modulo16bytes`,  
    `kEDMA_Modulo32bytes`,  
    `kEDMA_Modulo64bytes`,  
    `kEDMA_Modulo128bytes`,  
    `kEDMA_Modulo256bytes`,  
    `kEDMA_Modulo512bytes`,  
    `kEDMA_Modulo1Kbytes`,  
    `kEDMA_Modulo2Kbytes`,  
    `kEDMA_Modulo4Kbytes`,  
    `kEDMA_Modulo8Kbytes`,  
    `kEDMA_Modulo16Kbytes`,  
    `kEDMA_Modulo32Kbytes`,  
    `kEDMA_Modulo64Kbytes`,  
    `kEDMA_Modulo128Kbytes`,  
    `kEDMA_Modulo256Kbytes`,  
    `kEDMA_Modulo512Kbytes`,  
    `kEDMA_Modulo1Mbytes`,  
    `kEDMA_Modulo2Mbytes`,  
    `kEDMA_Modulo4Mbytes`,  
    `kEDMA_Modulo8Mbytes`,  
    `kEDMA_Modulo16Mbytes`,  
    `kEDMA_Modulo32Mbytes`,  
    `kEDMA_Modulo64Mbytes`,  
    `kEDMA_Modulo128Mbytes`,  
    `kEDMA_Modulo256Mbytes`,  
    `kEDMA_Modulo512Mbytes`,  
    `kEDMA_Modulo1Gbytes`,  
    `kEDMA_Modulo2Gbytes` }

*eDMA modulo configuration*

- enum `edma_bandwidth_t` {

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

Bandwidth control.
- enum `edma_channel_link_type_t` {

```
kEDMA_LinkNone = 0x0U,
kEDMA_MinorLink,
kEDMA_MajorLink }
```

Channel link type.
- enum `_edma_channel_status_flags` {

```
kEDMA_DoneFlag = 0x1U,
kEDMA_ErrorFlag = 0x2U,
kEDMA_InterruptFlag = 0x4U }
```

eDMA channel status flags.
- enum `_edma_error_status_flags` {

```
kEDMA_DestinationBusErrorFlag = DMA_ES_DBE_MASK,
kEDMA_SourceBusErrorFlag = DMA_ES_SBE_MASK,
kEDMA_ScatterGatherErrorFlag = DMA_ES_SGE_MASK,
kEDMA_NbytesErrorFlag = DMA_ES_NCE_MASK,
kEDMA_DestinationOffsetErrorFlag = DMA_ES_DOE_MASK,
kEDMA_DestinationAddressErrorFlag = DMA_ES_DAE_MASK,
kEDMA_SourceOffsetErrorFlag = DMA_ES_SOE_MASK,
kEDMA_SourceAddressErrorFlag = DMA_ES_SAE_MASK,
kEDMA_ErrorChannelFlag = DMA_ES_ERRCHN_MASK,
kEDMA_ChannelPriorityErrorFlag = DMA_ES_CPE_MASK,
kEDMA_TransferCanceledFlag = DMA_ES_ECX_MASK,
kEDMA_ValidFlag = (int)DMA_ES_VLD_MASK }
```

eDMA channel error status flags.
- enum `edma_interrupt_enable_t` {

```
kEDMA_ErrorInterruptEnable = 0x1U,
kEDMA_MajorInterruptEnable = DMA_CSR_INTMAJOR_MASK,
kEDMA_HalfInterruptEnable = DMA_CSR_INTHALF_MASK }
```

eDMA interrupt source
- enum `edma_transfer_type_t` {

```
kEDMA_MemoryToMemory = 0x0U,
kEDMA_PeripheralToMemory,
kEDMA_MemoryToPeripheral }
```

eDMA transfer type
- enum `_edma_transfer_status` {

```
kStatus_EDMA_QueueFull = MAKE_STATUS(kStatusGroup_EDMA, 0),
kStatus_EDMA_Busy = MAKE_STATUS(kStatusGroup_EDMA, 1) }
```

eDMA transfer status

Driver version

- #define `FSL_EDMA_DRIVER_VERSION` (MAKE_VERSION(2, 1, 4))
- eDMA driver version*

Typical use case

eDMA initialization and de-initialization

- void `EDMA_Init` (DMA_Type *base, const `edma_config_t` *config)
Initializes the eDMA peripheral.
- void `EDMA_Deinit` (DMA_Type *base)
Deinitializes the eDMA peripheral.
- void `EDMA_InstallTCD` (DMA_Type *base, uint32_t channel, `edma_tcd_t` *tcd)
Push content of TCD structure into hardware TCD register.
- void `EDMA_GetDefaultConfig` (`edma_config_t` *config)
Gets the eDMA default configuration structure.

eDMA Channel Operation

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

eDMA TCD Operation

- void `EDMA_TcdReset` (`edma_tcd_t` *tcd)
Sets all fields to default values for the TCD structure.
- void `EDMA_TcdSetTransferConfig` (`edma_tcd_t` *tcd, const `edma_transfer_config_t` *config, `edma_tcd_t` *nextTcd)
Configures the eDMA TCD transfer attribute.
- void `EDMA_TcdSetMinorOffsetConfig` (`edma_tcd_t` *tcd, const `edma_minor_offset_config_t` *config)
Configures the eDMA TCD minor offset feature.
- void `EDMA_TcdSetChannelLink` (`edma_tcd_t` *tcd, `edma_channel_link_type_t` type, uint32_t linkedChannel)

- static void **EDMA_TcdSetBandWidth** (**edma_tcd_t** *tcd, **edma_bandwidth_t** bandWidth)

Sets the channel link for the eDMA TCD.
- void **EDMA_TcdSetModulo** (**edma_tcd_t** *tcd, **edma_modulo_t** srcModulo, **edma_modulo_t** destModulo)

Sets the bandwidth for the eDMA TCD.
- static void **EDMA_TcdEnableAutoStopRequest** (**edma_tcd_t** *tcd, bool enable)

Sets the source modulo and the destination modulo for the eDMA TCD.
- void **EDMA_TcdEnableInterrupts** (**edma_tcd_t** *tcd, uint32_t mask)

Enables the auto stop request 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)

Data Structure Documentation

- void **EDMA_AbortTransfer** (edma_handle_t *handle)
eDMA stops transfer.
- static uint32_t **EDMA_GetUnusedTCDNumber** (edma_handle_t *handle)
Get unused TCD slot number.
- static uint32_t **EDMA_GetNextTCDAddress** (edma_handle_t *handle)
Get the next tcd address.
- void **EDMA_HandleIRQ** (edma_handle_t *handle)
eDMA IRQ handler for the current major loop transfer completion.

12.3 Data Structure Documentation

12.3.1 struct edma_config_t

Data Fields

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

12.3.1.0.0.25 Field Documentation

12.3.1.0.0.25.1 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.

12.3.1.0.0.25.2 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.

12.3.1.0.0.25.3 bool edma_config_t::enableDebugMode

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

12.3.2 struct edma_transfer_config_t

This structure configures the source/destination transfer attribute.

Data Fields

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

12.3.2.0.0.26 Field Documentation

12.3.2.0.0.26.1 `uint32_t edma_transfer_config_t::srcAddr`

12.3.2.0.0.26.2 `uint32_t edma_transfer_config_t::destAddr`

12.3.2.0.0.26.3 `edma_transfer_size_t edma_transfer_config_t::srcTransferSize`

12.3.2.0.0.26.4 `edma_transfer_size_t edma_transfer_config_t::destTransferSize`

12.3.2.0.0.26.5 `int16_t edma_transfer_config_t::srcOffset`

12.3.2.0.0.26.6 `int16_t edma_transfer_config_t::destOffset`

12.3.2.0.0.26.7 `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.

Data Structure Documentation

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.

12.3.4.0.0.27 Field Documentation

12.3.4.0.0.27.1 bool edma_minor_offset_config_t::enableSrcMinorOffset

12.3.4.0.0.27.2 bool edma_minor_offset_config_t::enableDestMinorOffset

12.3.4.0.0.27.3 uint32_t edma_minor_offset_config_t::minorOffset

12.3.5 struct edma_tcd_t

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

Data Fields

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

12.3.5.0.0.28 Field Documentation**12.3.5.0.0.28.1 __IO uint16_t edma_tcd_t::CITER****12.3.5.0.0.28.2 __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.

12.3.6.0.0.29 Field Documentation**12.3.6.0.0.29.1 edma_callback edma_handle_t::callback****12.3.6.0.0.29.2 void* edma_handle_t::userData****12.3.6.0.0.29.3 DMA_Type* edma_handle_t::base****12.3.6.0.0.29.4 edma_tcd_t* edma_handle_t::tcdPool****12.3.6.0.0.29.5 uint8_t edma_handle_t::channel****12.3.6.0.0.29.6 volatile int8_t edma_handle_t::header**

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

12.3.6.0.0.29.7 volatile int8_t edma_handle_t::tail

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

Typedef Documentation

12.3.6.0.0.29.8 volatile int8_t edma_handle_t::tcdUsed

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

12.3.6.0.0.29.9 volatile int8_t edma_handle_t::tcdSize

12.3.6.0.0.29.10 uint8_t edma_handle_t::flags

12.4 Macro Definition Documentation

12.4.1 #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))

Version 2.1.4.

12.5 Typedef Documentation

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

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

Parameters

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

<i>tcds</i>	How many tcds are done from the last callback. This parameter only used in scatter gather mode. It tells user how many tcds are finished between the last callback and this.
-------------	--

12.6 Enumeration Type Documentation

12.6.1 enum edma_transfer_size_t

Enumerator

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

12.6.2 enum edma_modulo_t

Enumerator

- kEDMA_ModuloDisable* Disable modulo.
- kEDMA_Modulo2bytes* Circular buffer size is 2 bytes.
- kEDMA_Modulo4bytes* Circular buffer size is 4 bytes.
- kEDMA_Modulo8bytes* Circular buffer size is 8 bytes.
- kEDMA_Modulo16bytes* Circular buffer size is 16 bytes.
- kEDMA_Modulo32bytes* Circular buffer size is 32 bytes.
- kEDMA_Modulo64bytes* Circular buffer size is 64 bytes.
- kEDMA_Modulo128bytes* Circular buffer size is 128 bytes.
- kEDMA_Modulo256bytes* Circular buffer size is 256 bytes.
- kEDMA_Modulo512bytes* Circular buffer size is 512 bytes.
- kEDMA_Modulo1Kbytes* Circular buffer size is 1 K bytes.
- kEDMA_Modulo2Kbytes* Circular buffer size is 2 K bytes.
- kEDMA_Modulo4Kbytes* Circular buffer size is 4 K bytes.
- kEDMA_Modulo8Kbytes* Circular buffer size is 8 K bytes.
- kEDMA_Modulo16Kbytes* Circular buffer size is 16 K bytes.
- kEDMA_Modulo32Kbytes* Circular buffer size is 32 K bytes.
- kEDMA_Modulo64Kbytes* Circular buffer size is 64 K bytes.
- kEDMA_Modulo128Kbytes* Circular buffer size is 128 K bytes.
- kEDMA_Modulo256Kbytes* Circular buffer size is 256 K bytes.
- kEDMA_Modulo512Kbytes* Circular buffer size is 512 K bytes.
- kEDMA_Modulo1Mbytes* Circular buffer size is 1 M bytes.
- kEDMA_Modulo2Mbytes* Circular buffer size is 2 M bytes.
- kEDMA_Modulo4Mbytes* Circular buffer size is 4 M bytes.

Enumeration Type Documentation

kEDMA_Modulo8Mbytes Circular buffer size is 8 M bytes.
kEDMA_Modulo16Mbytes Circular buffer size is 16 M bytes.
kEDMA_Modulo32Mbytes Circular buffer size is 32 M bytes.
kEDMA_Modulo64Mbytes Circular buffer size is 64 M bytes.
kEDMA_Modulo128Mbytes Circular buffer size is 128 M bytes.
kEDMA_Modulo256Mbytes Circular buffer size is 256 M bytes.
kEDMA_Modulo512Mbytes Circular buffer size is 512 M bytes.
kEDMA_Modulo1Gbytes Circular buffer size is 1 G bytes.
kEDMA_Modulo2Gbytes Circular buffer size is 2 G bytes.

12.6.3 enum edma_bandwidth_t

Enumerator

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

12.6.4 enum edma_channel_link_type_t

Enumerator

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

12.6.5 enum _edma_channel_status_flags

Enumerator

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

12.6.6 enum _edma_error_status_flags

Enumerator

kEDMA_DestinationBusErrorFlag Bus error on destination address.
kEDMA_SourceBusErrorFlag Bus error on the source address.

kEDMA_ScatterGatherErrorFlag Error on the Scatter/Gather address, not 32byte aligned.

kEDMA_NbytesErrorFlag NBYTES/CITER configuration error.

kEDMA_DestinationOffsetErrorFlag Destination offset not aligned with destination size.

kEDMA_DestinationAddressErrorFlag Destination address not aligned with destination size.

kEDMA_SourceOffsetErrorFlag Source offset not aligned with source size.

kEDMA_SourceAddressErrorFlag Source address not aligned with source size.

kEDMA_ErrorChannelFlag Error channel number of the cancelled channel number.

kEDMA_ChannelPriorityErrorFlag Channel priority is not unique.

kEDMA_TransferCanceledFlag Transfer cancelled.

kEDMA_ValidFlag No error occurred, this bit is 0. Otherwise, it is 1.

12.6.7 enum edma_interrupt_enable_t

Enumerator

kEDMA_ErrorInterruptEnable Enable interrupt while channel error occurs.

kEDMA_MajorInterruptEnable Enable interrupt while major count exhausted.

kEDMA_HalfInterruptEnable Enable interrupt while major count to half value.

12.6.8 enum edma_transfer_type_t

Enumerator

kEDMA_MemoryToMemory Transfer from memory to memory.

kEDMA_PeripheralToMemory Transfer from peripheral to memory.

kEDMA_MemoryToPeripheral Transfer from memory to peripheral.

12.6.9 enum _edma_transfer_status

Enumerator

kStatus_EDMA_QueueFull TCD queue is full.

kStatus_EDMA_Busy Channel is busy and can't handle the transfer request.

12.7 Function Documentation

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

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

Function Documentation

Parameters

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

Note

This function enables the minor loop map feature.

12.7.2 void EDMA_Deinit (DMA_Type * *base*)

This function gates the eDMA clock.

Parameters

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

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

Parameters

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

12.7.4 void EDMA_GetDefaultConfig (edma_config_t * *config*)

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

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

Parameters

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

12.7.5 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.

12.7.6 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.

12.7.7 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.

12.7.8 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.

12.7.9 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.

12.7.10 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

12.7.11 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.

12.7.12 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.

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.

Function Documentation

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.

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.

Function Documentation

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).

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.

Function Documentation

Parameters

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

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 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]

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.

Function Documentation

**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. This function shall only be used while users need to use scatter gather mode. Scatter gather mode enables EDMA to load a new transfer control block (tcd) in hardware, and automatically reconfigure that DMA channel for a new transfer. Users need to prepare tcd memory and also configure tcds using interface EDMA_SubmitTransfer.

Parameters

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

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. This function will be called every time one tcd finished transfer.

Parameters

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

12.7.34 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. In scatter gather mode, call this function will add a configured tcd to the circular list of tcd pool. The tcd pools is setup by call function EDMA_InstallTCDMemory before.

Parameters

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

Return values

<i>kStatus_EDMA_Success</i>	It means submit transfer request succeed.
<i>kStatus_EDMA_Queue-Full</i>	It means TCD queue is full. Submit transfer request is not allowed.

<i>kStatus_EDMA_Busy</i>	It means the given channel is busy, need to submit request later.
--------------------------	---

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

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

Function Documentation

Parameters

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

Returns

The unused tcd slot number.

12.7.40 static uint32_t EDMA_GetNextTCDAAddress (edma_handle_t * *handle*) [inline], [static]

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

Parameters

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

Returns

The next TCD address.

12.7.41 void EDMA_HandleIRQ (edma_handle_t * *handle*)

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

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

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

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

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

Parameters

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

Function Documentation

Chapter 13

ENET: Ethernet MAC Driver

13.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 10/100 Mbps Ethernet MAC (ENET) module of MCUXpresso SDK 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 the 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.

This group has the receive active API [ENET_ActiveRead\(\)](#) and [ENET_ActiveReadMultiRing\(\)](#) for single and multiple rings. The [ENET_AVBConfigure\(\)](#) is provided to configure the AVB features to support the AVB frames transmission. Note that due to the AVB frames transmission scheme being a credit-based TX scheme, it is only supported with the Enhanced buffer descriptors. Because of this, the AVB configuration should only be done with the Enhanced buffer descriptor. When the AVB feature is required, make sure the the "ENET_ENHANCEDBUFFERDESCRIPTOR_MODE" is defined before using this feature.

1. For single ring

For ENET receive, the [ENET_GetRxFrameSize\(\)](#) function needs to 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.

1. For multiple-ring supported

The ENET driver now added a series transactional APIs with postfix "MultiRing" to support the extended multiple-ring for AVB feature. There are extended multiple-ring functions for receive side: [ENET_GetRxErrBeforeReadFrameMultiRing\(\)](#), [ENET_GetRxFrameSizeMultiRing\(\)](#), and [ENET_ReadFrameMultiRing\(\)](#). They are the similar to the single ring receive APIs and only add the "ringId" input param to identify the different ring index. For TX side add the [ENET_SendFrameMultiRing](#), [ENET_GetTxErr-](#)

Typical use case

AfterSendFrameMultiRing(). They are similar to the single ring transmit APIs and only add the "ring-Id" input param to identify the different ring index.

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 needs to 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. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/enet 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. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/enet

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 statistics structure. [More...](#)
- struct [enet_buffer_config_t](#)
Defines the receive buffer descriptor configuration structure. [More...](#)
- struct [enet_ptp_time_t](#)
Defines the ENET PTP time stamp structure. [More...](#)
- struct [enet_ptp_time_data_t](#)
Defines the structure for the ENET PTP message data and timestamp data. [More...](#)
- struct [enet_ptp_time_data_ring_t](#)
Defines the ENET PTP ring buffer structure for the PTP message timestamp store. [More...](#)
- struct [enet_ptp_config_t](#)
Defines the ENET PTP 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.

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_TxFrameOverLen = MAKE_STATUS(kStatusGroup_ENET, 3U),`
 `kStatus_ENET_TxFrameBusy = MAKE_STATUS(kStatusGroup_ENET, 4U),`
 `kStatus_ENET_TxFrameFail = MAKE_STATUS(kStatusGroup_ENET, 5U),`
 `kStatus_ENET_PtpTsRingFull = MAKE_STATUS(kStatusGroup_ENET, 6U),`
 `kStatus_ENET_PtpTsRingEmpty = MAKE_STATUS(kStatusGroup_ENET, 7U) }`
Defines the status return codes for transaction.
- `enum enet_mii_mode_t {`
 `kENET_MiiMode = 0U,`
 `kENET_RmiiMode = 1U }`
Defines the MII/RMII/RGMII mode for data interface between the MAC and the PHY.
- `enum enet_mii_speed_t {`
 `kENET_MiiSpeed10M = 0U,`
 `kENET_MiiSpeed100M = 1U }`
Defines the 10/100/1000 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 }`
Define the MII opcode for normal MDIO_CLAUSES_22 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,`

Typical use case

```
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_EBusERIInterrupt = 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,
 kENET_TsAvailInterrupt = ENET_EIR_TS_AVAIL_MASK,
 kENET_TsTimerInterrupt = ENET_EIR_TS_TIMER_MASK }

List of interrupts supported by the peripheral.

- enum `enet_event_t` {
 kENET_RxEvent,
 kENET_TxEvent,
 kENET_ErrEvent,
 kENET_WakeUpEvent,
 kENET_TimeStampEvent,
 kENET_TimeStampAvailEvent }

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.

- enum `enet_ptp_event_type_t` {
 kENET_PtpEventMsgType = 3U,
 kENET_PtpSrcPortIdLen = 10U,
 kENET_PtpEventPort = 319U,
 kENET_PtpGnrlPort = 320U }

Defines the ENET PTP message related constant.

- enum `enet_ptp_timer_channel_t` {

kENET_PtpTimerChannel1 = 0U,

kENET_PtpTimerChannel2,

kENET_PtpTimerChannel3,

kENET_PtpTimerChannel4 }

Defines the IEEE 1588 PTP timer channel numbers.

- enum `enet_ptp_timer_channel_mode_t` {

kENET_PtpChannelDisable = 0U,

kENET_PtpChannelRisingCapture = 1U,

kENET_PtpChannelFallingCapture = 2U,

kENET_PtpChannelBothCapture = 3U,

kENET_PtpChannelSoftCompare = 4U,

kENET_PtpChannelToggleCompare = 5U,

kENET_PtpChannelClearCompare = 6U,

kENET_PtpChannelSetCompare = 7U,

kENET_PtpChannelClearCompareSetOverflow = 10U,

kENET_PtpChannelSetCompareClearOverflow = 11U,

kENET_PtpChannelPulseLowonCompare = 14U,

kENET_PtpChannelPulseHighonCompare = 15U }

Defines the capture or compare mode for IEEE 1588 PTP timer channels.

Driver version

- #define `FSL_ENET_DRIVER_VERSION` (MAKE_VERSION(2, 2, 3))
Defines the driver version.

ENET DESCRIPTOR QUEUE

- #define `FSL_FEATURE_ENET_QUEUE` 1 /* Singal queue for previous IP. */
Defines the queue number.

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

Typical use case

- `#define ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_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_SOFTWENER1_MASK 0x4000U`
Software owner one mask.
- `#define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U`
Wrap buffer descriptor mask.
- `#define ENET_BUFFDESCRIPTOR_TX_SOFTWENER2_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.

First extended control region bit masks of the receive buffer descriptor.

- `#define ENET_BUFFDESCRIPTOR_RX_IPV4_MASK 0x0001U`
Ipv4 frame mask.
- `#define ENET_BUFFDESCRIPTOR_RX_IPV6_MASK 0x0002U`
Ipv6 frame mask.
- `#define ENET_BUFFDESCRIPTOR_RX_VLAN_MASK 0x0004U`
VLAN frame mask.
- `#define ENET_BUFFDESCRIPTOR_RX_PROTOCOLCHECKSUM_MASK 0x0010U`
Protocol checksum error mask.
- `#define ENET_BUFFDESCRIPTOR_RX_IPHEADCHECKSUM_MASK 0x0020U`
IP header checksum error mask.

Second extended control region bit masks of the receive buffer descriptor.

- `#define ENET_BUFFDESCRIPTOR_RX_INTERRUPT_MASK 0x0080U`
BD interrupt mask.
- `#define ENET_BUFFDESCRIPTOR_RX_UNICAST_MASK 0x0100U`
Unicast frame mask.
- `#define ENET_BUFFDESCRIPTOR_RX_COLLISION_MASK 0x0200U`
BD collision mask.
- `#define ENET_BUFFDESCRIPTOR_RX_PHYERR_MASK 0x0400U`
PHY error mask.
- `#define ENET_BUFFDESCRIPTOR_RX_MACERR_MASK 0x8000U`

Mac error mask.

First extended control region bit masks of the transmit buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_TX_ERR_MASK 0x8000U
Transmit error mask.
- #define ENET_BUFFDESCRIPTOR_TX_UNDERFLOWERR_MASK 0x2000U
Underflow error mask.
- #define ENET_BUFFDESCRIPTOR_TX_EXCCOLLISIONERR_MASK 0x1000U
Excess collision error mask.
- #define ENET_BUFFDESCRIPTOR_TX_FRAMEERR_MASK 0x0800U
Frame error mask.
- #define ENET_BUFFDESCRIPTOR_TX_LATECOLLISIONERR_MASK 0x0400U
Late collision error mask.
- #define ENET_BUFFDESCRIPTOR_TX_OVERFLOWERR_MASK 0x0200U
Overflow error mask.
- #define ENET_BUFFDESCRIPTOR_TX_TIMESTAMPERR_MASK 0x0100U
Timestamp error mask.

Second extended control region bit masks of the transmit buffer descriptor.

- #define ENET_BUFFDESCRIPTOR_TX_INTERRUPT_MASK 0x4000U
Interrupt mask.
- #define ENET_BUFFDESCRIPTOR_TX_TIMESTAMP_MASK 0x2000U
Timestamp flag mask.

Defines some Ethernet parameters.

- #define ENET_FRAME_MAX_FRAMELEN 1518U
Default maximum Ethernet frame size.
- #define ENET_FIFO_MIN_RX_FULL 5U
ENET minimum receive FIFO full.
- #define ENET_RX_MIN_BUFFERSIZE 256U
ENET minimum buffer size.
- #define ENET_PHY_MAXADDRESS (ENET_MMFR_PA_MASK >> ENET_MMFR_PA_SHIFT)
- #define ENET_TX_INTERRUPT (kENET_TxFrameInterrupt | kENET_TxBufferInterrupt)
- #define ENET_RX_INTERRUPT (kENET_RxFrameInterrupt | kENET_RxBufferInterrupt)
- #define ENET_TS_INTERRUPT (kENET_TsTimerInterrupt | kENET_TsAvailInterrupt)
- #define ENET_ERR_INTERRUPT
- #define ENET_ERR_INTERRUPT

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.

Typical use case

- 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.
- 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 operation

- 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 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](#) *eError-Static)
Gets the error statistics of a received frame for ENET single ring.
- status_t [ENET_GetTxErrAfterSendFrame](#) (enet_handle_t *handle, [enet_data_error_stats_t](#) *eError-Static)
Gets the ENET transmit frame statistics after the data send for single ring.
- status_t [ENET_GetRxFrameSize](#) (enet_handle_t *handle, uint32_t *length)
Gets the size of the read frame for single ring.
- status_t [ENET_ReadFrame](#) (ENET_Type *base, enet_handle_t *handle, uint8_t *data, uint32_t length)
Reads a frame from the ENET device for single ring.
- status_t [ENET_SendFrame](#) (ENET_Type *base, enet_handle_t *handle, const uint8_t *data, uint32_t length)
Transmits an ENET frame for single ring.
- 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)
Some special IRQ handler including the error, mii, wakeup irq handler.
- void [ENET_CommonFrame0IRQHandler](#) (ENET_Type *base)
the common IRQ handler for the tx/rx/error etc irq handler.

ENET PTP 1588 function operation

- void [ENET_Ptp1588Configure](#) (ENET_Type *base, enet_handle_t *handle, [enet_ptp_config_t](#) *ptpConfig)
Configures the ENET PTP IEEE 1588 feature with the basic configuration.
- void [ENET_Ptp1588StartTimer](#) (ENET_Type *base, uint32_t ptptClkSrc)
Starts the ENET PTP 1588 Timer.
- static void [ENET_Ptp1588StopTimer](#) (ENET_Type *base)
Stops the ENET PTP 1588 Timer.
- void [ENET_Ptp1588AdjustTimer](#) (ENET_Type *base, uint32_t corrIncrease, uint32_t corrPeriod)
Adjusts the ENET PTP 1588 timer.
- static void [ENET_Ptp1588SetChannelMode](#) (ENET_Type *base, [enet_ptp_timer_channel_t](#) channel, [enet_ptp_timer_channel_mode_t](#) mode, bool intEnable)
Sets the ENET PTP 1588 timer channel mode.
- static void [ENET_Ptp1588SetChannelCmpValue](#) (ENET_Type *base, [enet_ptp_timer_channel_t](#) channel, uint32_t cmpValue)
Sets the ENET PTP 1588 timer channel comparison value.
- static bool [ENET_Ptp1588GetChannelStatus](#) (ENET_Type *base, [enet_ptp_timer_channel_t](#) channel)
Gets the ENET PTP 1588 timer channel status.
- static void [ENET_Ptp1588ClearChannelStatus](#) (ENET_Type *base, [enet_ptp_timer_channel_t](#) channel)
Clears the ENET PTP 1588 timer channel status.

Data Structure Documentation

- void [ENET_Ptp1588GetTimer](#) (ENET_Type *base, enet_handle_t *handle, [enet_ptp_time_t](#) *ptpTime)
Gets the current ENET time from the PTP 1588 timer.
- void [ENET_Ptp1588SetTimer](#) (ENET_Type *base, enet_handle_t *handle, [enet_ptp_time_t](#) *ptpTime)
Sets the ENET PTP 1588 timer to the assigned time.
- void [ENET_Ptp1588TimerIRQHandler](#) (ENET_Type *base, enet_handle_t *handle)
The IEEE 1588 PTP time stamp interrupt handler.
- status_t [ENET_GetRxFrameTime](#) (enet_handle_t *handle, [enet_ptp_time_data_t](#) *ptpTimeData)
Gets the time stamp of the received frame.
- status_t [ENET_GetTxFrameTime](#) (enet_handle_t *handle, [enet_ptp_time_data_t](#) *ptpTimeData)
Gets the time stamp of the transmit frame.

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.
- uint16_t **controlExtend0**
Extend buffer descriptor control0.
- uint16_t **controlExtend1**
Extend buffer descriptor control1.
- uint16_t **payloadCheckSum**
Internal payload checksum.
- uint8_t **headerLength**
Header length.
- uint8_t **protocolType**
Protocol type.
- uint16_t **controlExtend2**
Extend buffer descriptor control2.
- uint32_t **timestamp**
Timestamp.

13.3.1.0.0.30 Field Documentation

- 13.3.1.0.0.30.1 `uint16_t enet_rx_bd_struct_t::length`
- 13.3.1.0.0.30.2 `uint16_t enet_rx_bd_struct_t::control`
- 13.3.1.0.0.30.3 `uint8_t* enet_rx_bd_struct_t::buffer`
- 13.3.1.0.0.30.4 `uint16_t enet_rx_bd_struct_t::controlExtend0`
- 13.3.1.0.0.30.5 `uint16_t enet_rx_bd_struct_t::controlExtend1`
- 13.3.1.0.0.30.6 `uint16_t enet_rx_bd_struct_t::payloadCheckSum`
- 13.3.1.0.0.30.7 `uint8_t enet_rx_bd_struct_t::headerLength`
- 13.3.1.0.0.30.8 `uint8_t enet_rx_bd_struct_t::protocolType`
- 13.3.1.0.0.30.9 `uint16_t enet_rx_bd_struct_t::controlExtend2`
- 13.3.1.0.0.30.10 `uint32_t enet_rx_bd_struct_t::timestamp`

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.
- `uint16_t controlExtend0`
Extend buffer descriptor control0.
- `uint16_t controlExtend1`
Extend buffer descriptor control1.
- `uint16_t controlExtend2`
Extend buffer descriptor control2.
- `uint32_t timestamp`
Timestamp.

Data Structure Documentation

13.3.2.0.0.31 Field Documentation

- 13.3.2.0.0.31.1 `uint16_t enet_tx_bd_struct_t::length`
- 13.3.2.0.0.31.2 `uint16_t enet_tx_bd_struct_t::control`
- 13.3.2.0.0.31.3 `uint8_t* enet_tx_bd_struct_t::buffer`
- 13.3.2.0.0.31.4 `uint16_t enet_tx_bd_struct_t::controlExtend0`
- 13.3.2.0.0.31.5 `uint16_t enet_tx_bd_struct_t::controlExtend1`
- 13.3.2.0.0.31.6 `uint16_t enet_tx_bd_struct_t::controlExtend2`
- 13.3.2.0.0.31.7 `uint32_t enet_tx_bd_struct_t::timestamp`

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.
- `uint32_t statsRxProtocolChecksumErr`
Receive protocol checksum error.
- `uint32_t statsRxIpHeadChecksumErr`
Receive IP header checksum error.
- `uint32_t statsRxMacErr`
Receive Mac error.
- `uint32_t statsRxPhyErr`
Receive PHY error.
- `uint32_t statsRxCollisionErr`
Receive collision.
- `uint32_t statsTxErr`
The error happen when transmit the frame.
- `uint32_t statsTxFramesErr`
The transmit frame is error.
- `uint32_t statsTxOverflowErr`
Transmit overflow.
- `uint32_t statsTxLateCollisionErr`
Transmit late collision.
- `uint32_t statsTxExcessCollisionErr`
Transmit excess collision.

- `uint32_t statsTxUnderFlowErr`
Transmit under flow error.
- `uint32_t statsTxTsErr`
Transmit time stamp error.

13.3.3.0.0.32 Field Documentation

- 13.3.3.0.0.32.1 `uint32_t enet_data_error_stats_t::statsRxLenGreaterErr`
- 13.3.3.0.0.32.2 `uint32_t enet_data_error_stats_t::statsRxFcsErr`
- 13.3.3.0.0.32.3 `uint32_t enet_data_error_stats_t::statsRxOverRunErr`
- 13.3.3.0.0.32.4 `uint32_t enet_data_error_stats_t::statsRxTruncateErr`
- 13.3.3.0.0.32.5 `uint32_t enet_data_error_stats_t::statsRxProtocolChecksumErr`
- 13.3.3.0.0.32.6 `uint32_t enet_data_error_stats_t::statsRxIpHeadChecksumErr`
- 13.3.3.0.0.32.7 `uint32_t enet_data_error_stats_t::statsRxMacErr`
- 13.3.3.0.0.32.8 `uint32_t enet_data_error_stats_t::statsRxPhyErr`
- 13.3.3.0.0.32.9 `uint32_t enet_data_error_stats_t::statsRxCollisionErr`
- 13.3.3.0.0.32.10 `uint32_t enet_data_error_stats_t::statsTxErr`
- 13.3.3.0.0.32.11 `uint32_t enet_data_error_stats_t::statsTxFrameErr`
- 13.3.3.0.0.32.12 `uint32_t enet_data_error_stats_t::statsTxOverFlowErr`
- 13.3.3.0.0.32.13 `uint32_t enet_data_error_stats_t::statsTxLateCollisionErr`
- 13.3.3.0.0.32.14 `uint32_t enet_data_error_stats_t::statsTxExcessCollisionErr`
- 13.3.3.0.0.32.15 `uint32_t enet_data_error_stats_t::statsTxUnderFlowErr`
- 13.3.3.0.0.32.16 `uint32_t enet_data_error_stats_t::statsTxTsErr`

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 ENET_BUFF_ALIGNMENT. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of "ENET_BUFF_ALIGNMENT" and the cache line size.
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 ENET_BUFF_ALIGNMENT. buffer descriptors should be put in non-cacheable region when cache is enabled.
3. The aligned transmit and receive data buffer start address must be evenly divisible by ENET_BU-

Data Structure Documentation

F_ALIGNMENT. 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". when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of "ENET_BUFF_ALIGNMENT" and the cache line size.

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: should be non-cacheable.
- `volatile enet_tx_bd_struct_t * txBdStartAddrAlign`
Aligned transmit buffer descriptor start address: should be non-cacheable.
- `uint8_t * rxBufferAlign`
Receive data buffer start address.
- `uint8_t * txBufferAlign`
Transmit data buffer start address.

13.3.4.0.0.33 Field Documentation

13.3.4.0.0.33.1 `uint16_t enet_buffer_config_t::rxBdNumber`

13.3.4.0.0.33.2 `uint16_t enet_buffer_config_t::txBdNumber`

13.3.4.0.0.33.3 `uint32_t enet_buffer_config_t::rxBuffSizeAlign`

13.3.4.0.0.33.4 `uint32_t enet_buffer_config_t::txBuffSizeAlign`

13.3.4.0.0.33.5 `volatile enet_rx_bd_struct_t* enet_buffer_config_t::rxBdStartAddrAlign`

13.3.4.0.0.33.6 `volatile enet_tx_bd_struct_t* enet_buffer_config_t::txBdStartAddrAlign`

13.3.4.0.0.33.7 `uint8_t* enet_buffer_config_t::rxBufferAlign`

13.3.4.0.0.33.8 `uint8_t* enet_buffer_config_t::txBufferAlign`

13.3.5 `struct enet_ptp_time_t`

Data Fields

- `uint64_t second`

- `uint32_t nanosecond`
Nanosecond.

13.3.5.0.0.34 Field Documentation

13.3.5.0.0.34.1 `uint64_t enet_ptp_time_t::second`

13.3.5.0.0.34.2 `uint32_t enet_ptp_time_t::nanosecond`

13.3.6 struct enet_ptp_time_data_t

Data Fields

- `uint8_t version`
PTP version.
- `uint8_t sourcePortId [kENET_PtpSrcPortIdLen]`
PTP source port ID.
- `uint16_t sequenceId`
PTP sequence ID.
- `uint8_t messageType`
PTP message type.
- `enet_ptp_time_t timeStamp`
PTP timestamp.

13.3.6.0.0.35 Field Documentation

13.3.6.0.0.35.1 `uint8_t enet_ptp_time_data_t::version`

13.3.6.0.0.35.2 `uint8_t enet_ptp_time_data_t::sourcePortId[kENET_PtpSrcPortIdLen]`

13.3.6.0.0.35.3 `uint16_t enet_ptp_time_data_t::sequenceId`

13.3.6.0.0.35.4 `uint8_t enet_ptp_time_data_t::messageType`

13.3.6.0.0.35.5 `enet_ptp_time_t enet_ptp_time_data_t::timeStamp`

13.3.7 struct enet_ptp_time_data_ring_t

Data Fields

- `uint32_t front`
The first index of the ring.
- `uint32_t end`
The end index of the ring.
- `uint32_t size`
The size of the ring.
- `enet_ptp_time_data_t * ptpTsData`
PTP message data structure.

Data Structure Documentation

13.3.7.0.0.36 Field Documentation

13.3.7.0.0.36.1 `uint32_t enet_ptp_time_data_ring_t::front`

13.3.7.0.0.36.2 `uint32_t enet_ptp_time_data_ring_t::end`

13.3.7.0.0.36.3 `uint32_t enet_ptp_time_data_ring_t::size`

13.3.7.0.0.36.4 `enet_ptp_time_data_t* enet_ptp_time_data_ring_t::ptpTsData`

13.3.8 struct `enet_ptp_config_t`

Data Fields

- `uint8_t ptpRxBuffNum`
Receive 1588 timestamp buffer number.
- `uint8_t ptpTxBuffNum`
Transmit 1588 timestamp buffer number.
- `enet_ptp_time_data_t * rxPtpTsData`
The start address of 1588 receive timestamp buffers.
- `enet_ptp_time_data_t * txPtpTsData`
The start address of 1588 transmit timestamp buffers.
- `enet_ptp_timer_channel_t channel`
Used for ERRATA_2579: the PTP 1588 timer channel for time interrupt.
- `uint32_t ptp1588ClockSrc_Hz`
The clock source of the PTP 1588 timer.

13.3.8.0.0.37 Field Documentation

13.3.8.0.0.37.1 `enet_ptp_timer_channel_t enet_ptp_config_t::channel`

13.3.8.0.0.37.2 `uint32_t enet_ptp_config_t::ptp1588ClockSrc_Hz`

13.3.9 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 pauseDuration, 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.
7. The intCoalesceCfg can be used in the rx or tx enabled cases to decrease the CPU loading.

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.
- **uint8_t ringNum**
Number of used rings.

Data Structure Documentation

13.3.9.0.0.38 Field Documentation

13.3.9.0.0.38.1 uint32_t enet_config_t::macSpecialConfig

A logical OR of "enet_special_control_flag_t".

13.3.9.0.0.38.2 uint32_t enet_config_t::interrupt

A logical OR of "enet_interrupt_enable_t".

13.3.9.0.0.38.3 uint16_t enet_config_t::rxMaxFrameLen

13.3.9.0.0.38.4 enet_mii_mode_t enet_config_t::miiMode

13.3.9.0.0.38.5 enet_mii_speed_t enet_config_t::miiSpeed

13.3.9.0.0.38.6 enet_mii_duplex_t enet_config_t::miiDuplex

13.3.9.0.0.38.7 uint8_t enet_config_t::rxAccelerConfig

13.3.9.0.0.38.8 uint8_t enet_config_t::txAccelerConfig

13.3.9.0.0.38.9 uint16_t enet_config_t::pauseDuration

13.3.9.0.0.38.10 uint8_t enet_config_t::rxFifoEmptyThreshold

13.3.9.0.0.38.11 uint8_t enet_config_t::rxFifoStatEmptyThreshold

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

13.3.9.0.0.38.12 uint8_t enet_config_t::rxFifoFullThreshold

13.3.9.0.0.38.13 uint8_t enet_config_t::txFifoWatermark

13.3.9.0.0.38.14 uint8_t enet_config_t::ringNum

default with 1 – single ring.

13.3.10 struct _enet_handle

Data Fields

- volatile [enet_rx_bd_struct_t * rxBdBase](#) [FSL_FEATURE_ENET_QUEUE]
Receive buffer descriptor base address pointer.
- volatile [enet_rx_bd_struct_t * rxBdCurrent](#) [FSL_FEATURE_ENET_QUEUE]
The current available receive buffer descriptor pointer.
- volatile [enet_tx_bd_struct_t * txBdBase](#) [FSL_FEATURE_ENET_QUEUE]
Transmit buffer descriptor base address pointer.
- volatile [enet_tx_bd_struct_t * txBdCurrent](#) [FSL_FEATURE_ENET_QUEUE]

- **The current available transmit buffer descriptor pointer.**
• `uint32_t rxBuffSizeAlign [FSL_FEATURE_ENET_QUEUE]`
Receive buffer size alignment.
- `uint32_t txBuffSizeAlign [FSL_FEATURE_ENET_QUEUE]`
Transmit buffer size alignment.
- `uint8_t ringNum`
Number of used rings.
- `enet_callback_t callback`
Callback function.
- `void * userData`
Callback function parameter.
- `volatile enet_tx_bd_struct_t * txBdDirtyStatic [FSL_FEATURE_ENET_QUEUE]`
The dirty transmit buffer descriptor for error static update.
- `volatile enet_tx_bd_struct_t * txBdDirtyTime [FSL_FEATURE_ENET_QUEUE]`
The dirty transmit buffer descriptor for time stamp update.
- `uint64_t msTimerSecond`
The second for Master PTP timer .
- `enet_ptp_time_data_ring_t rxPtpTsDataRing`
Receive PTP 1588 time stamp data ring buffer.
- `enet_ptp_time_data_ring_t txPtpTsDataRing`
Transmit PTP 1588 time stamp data ring buffer.

Macro Definition Documentation

13.3.10.0.0.39 Field Documentation

- 13.3.10.0.0.39.1 `volatile enet_rx_bd_struct_t* enet_handle_t::rxBdBase[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.2 `volatile enet_rx_bd_struct_t* enet_handle_t::rxBdCurrent[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.3 `volatile enet_tx_bd_struct_t* enet_handle_t::txBdBase[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.4 `volatile enet_tx_bd_struct_t* enet_handle_t::txBdCurrent[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.5 `uint32_t enet_handle_t::rxBuffSizeAlign[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.6 `uint32_t enet_handle_t::txBuffSizeAlign[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.7 `uint8_t enet_handle_t::ringNum`
- 13.3.10.0.0.39.8 `enet_callback_t enet_handle_t::callback`
- 13.3.10.0.0.39.9 `void* enet_handle_t::userData`
- 13.3.10.0.0.39.10 `volatile enet_tx_bd_struct_t* enet_handle_t::txBdDirtyStatic[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.11 `volatile enet_tx_bd_struct_t* enet_handle_t::txBdDirtyTime[FSL_FEATURE_ENET_QUEUE]`
- 13.3.10.0.0.39.12 `uint64_t enet_handle_t::msTimerSecond`
- 13.3.10.0.0.39.13 `enet_ptp_time_data_ring_t enet_handle_t::rxPtpTsDataRing`
- 13.3.10.0.0.39.14 `enet_ptp_time_data_ring_t enet_handle_t::txPtpTsDataRing`

13.4 Macro Definition Documentation

13.4.1 `#define FSL_ENET_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))`

Version 2.2.3.

Macro Definition Documentation

- 13.4.2 #define FSL_FEATURE_ENET_QUEUE 1 /* Singal queue for previous IP. */
- 13.4.3 #define ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK 0x8000U
- 13.4.4 #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK 0x4000U
- 13.4.5 #define ENET_BUFFDESCRIPTOR_RX_WRAP_MASK 0x2000U
- 13.4.6 #define ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask 0x1000U
- 13.4.7 #define ENET_BUFFDESCRIPTOR_RX_LAST_MASK 0x0800U
- 13.4.8 #define ENET_BUFFDESCRIPTOR_RX_MISS_MASK 0x0100U
- 13.4.9 #define ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK 0x0080U
- 13.4.10 #define ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK 0x0040U
- 13.4.11 #define ENET_BUFFDESCRIPTOR_RX_LENVLIOLATE_MASK 0x0020U
- 13.4.12 #define ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK 0x0010U
- 13.4.13 #define ENET_BUFFDESCRIPTOR_RX_CRC_MASK 0x0004U
- 13.4.14 #define ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK 0x0002U
- 13.4.15 #define ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK 0x0001U
- 13.4.16 #define ENET_BUFFDESCRIPTOR_TX_READY_MASK 0x8000U
- 13.4.17 #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK 0x4000U
- 13.4.18 #define ENET_BUFFDESCRIPTOR_TX_WRAP_MASK 0x2000U
- 13.4.19 #define ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK 0x1000U
- 13.4.20 #define ENET_BUFFDESCRIPTOR_TX_LAST_MASK 0x0800U
- 13.4.21 #define ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK 0x0400U
- 13.4.22 #define ENET_BUFFDESCRIPTOR_RX_IPV4_MASK 0x0001U

```
(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)
```

13.4.42 #define ENET_FRAME_MAX_FRAMELEN 1518U

13.4.43 #define ENET_FIFO_MIN_RX_FULL 5U

13.4.44 #define ENET_RX_MIN_BUFFERSIZE 256U

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_TxFrameOverLen Tx frame over length.

kStatus_ENET_TxFrameBusy Tx buffer descriptors are under process.

kStatus_ENET_TxFrameFail Transmit frame fail.

kStatus_ENET_PtpTsRingFull Timestamp ring full.

kStatus_ENET_PtpTsRingEmpty Timestamp ring empty.

13.6.2 enum enet_mii_mode_t

Enumerator

kENET_MiiMode MII mode for data interface.

kENET_RmiiMode RMII mode for data interface.

13.6.3 enum enet_mii_speed_t

Notice: "kENET_MiiSpeed1000M" only supported when mii mode is "kENET_RgmiiMode".

Enumeration Type Documentation

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 normal VLAN (single vlan tag).

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 error interrupt source.
kENET_WakeupInterrupt WAKEUP interrupt source.
kENET_TsAvailInterrupt TS AVAIL interrupt source for PTP.
kENET_TsTimerInterrupt TS WRAP interrupt source for PTP.

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.
kENET_TimeStampEvent Time stamp event.
kENET_TimeStampAvailEvent Time stamp available event.

Enumeration Type 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.

13.6.12 enum enet_ptp_event_type_t

Enumerator

- kENET_PtpEventMsgType* PTP event message type.
- kENET_PtpSrcPortIdLen* PTP message sequence id length.
- kENET_PtpEventPort* PTP event port number.
- kENET_PtpGnrlPort* PTP general port number.

13.6.13 enum enet_ptp_timer_channel_t

Enumerator

- kENET_PtpTimerChannel1* IEEE 1588 PTP timer Channel 1.
- kENET_PtpTimerChannel2* IEEE 1588 PTP timer Channel 2.
- kENET_PtpTimerChannel3* IEEE 1588 PTP timer Channel 3.
- kENET_PtpTimerChannel4* IEEE 1588 PTP timer Channel 4.

13.6.14 enum enet_ptp_timer_channel_mode_t

Enumerator

- kENET_PtpChannelDisable* Disable timer channel.

kENET_PtpChannelRisingCapture Input capture on rising edge.
kENET_PtpChannelFallingCapture Input capture on falling edge.
kENET_PtpChannelBothCapture Input capture on both edges.
kENET_PtpChannelSoftCompare Output compare software only.
kENET_PtpChannelToggleCompare Toggle output on compare.
kENET_PtpChannelClearCompare Clear output on compare.
kENET_PtpChannelSetCompare Set output on compare.
kENET_PtpChannelClearCompareSetOverflow Clear output on compare, set output on overflow.
kENET_PtpChannelSetCompareClearOverflow Set output on compare, clear output on overflow.
kENET_PtpChannelPulseLowonCompare Pulse output low on compare for one IEEE 1588 clock cycle.
kENET_PtpChannelPulseHighonCompare Pulse output high on compare for one IEEE 1588 clock cycle.

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 configure structure for [ENET_Init\(\)](#). User may use the initialized structure unchanged in [ENET_Init\(\)](#), or modify some fields of the structure before calling [ENET_Init\(\)](#). 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.
-------------	-------------------------------

Function Documentation

<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. It is the start address of "ringNum" enet_buffer_config structures. To support added multi-ring features in some soc and compatible with the previous enet driver version. For single ring supported, this bufferConfig is a buffer configure structure pointer, for multi-ring supported and used case, this bufferConfig pointer should be a buffer configure structure array pointer.
<i>macAddr</i>	ENET mac address of 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 reset state. Note that this function sets all registers to reset state. As a result, the ENET module can't work after calling this function.

Parameters

<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*)

This API is provided to dynamically change the speed and duplex for MAC.

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

Function Documentation

<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]

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*)

Used for standard IEEE802.3 MDIO Clause 22 format.

Parameters

<i>base</i>	ENET peripheral base address.
<i>phyAddr</i>	The PHY address.
<i>phyReg</i>	The PHY register. Range from 0 ~ 31.
<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*)

Used for standard IEEE802.3 MDIO Clause 22 format.

Parameters

<i>base</i>	ENET peripheral base address.
<i>phyAddr</i>	The PHY address.
<i>phyReg</i>	The PHY register. Range from 0 ~ 31.
<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*)

Function Documentation

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]

This function is to active the enet read process. It is used for single descriptor ring/queue.

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 to be used to as the mask to get the exact the accelerator option.
<i>rxAccelOption</i>	The receive accelerator option. The "enet_rx_accelerator_t" is recommended to be used to 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);
```

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

Function Documentation

<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);
*     }
*
```

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_GetTxErrAfterSendFrame (*enet_handle_t * handle*, *enet_data_error_stats_t * eErrorStatic*)

This interface gets the error statistics of the transmit frame. Because the error information is reported by the uDMA after the data delivery, this interface should be called after the data transmit API. It is recommended to call this function on transmit interrupt handler. After calling the ENET_SendFrame, the transmit interrupt notifies the transmit completion.

Parameters

Function Documentation

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

Returns

The execute status.

13.7.25 **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.

13.7.26 **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.

13.7.27 **status_t ENET_SendFrame (ENET_Type * *base*, enet_handle_t * *handle*, const 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> .

13.7.28 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.

13.7.29 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.

13.7.30 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.

13.7.31 void ENET_CommonFrame0IRQHandler (ENET_Type * *base*)

This is used for the combined tx/rx/error interrupt for single/mutli-ring (frame 0).

Parameters

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

13.7.32 void ENET_Ptp1588Configure (ENET_Type * *base*, enet_handle_t * *handle*, enet_ptp_config_t * *ptpConfig*)

The function sets the clock for PTP 1588 timer and enables time stamp interrupts and transmit interrupts for PTP 1588 features. This API should be called when the 1588 feature is enabled or the ENET_ENHANCEDBUFFERDESCRIPTOR_MODE is defined. ENET_Init should be called before calling this API.

Note

The PTP 1588 time-stamp second increase though time-stamp interrupt handler and the transmit time-stamp store is done through transmit interrupt handler. As a result, the TS interrupt and TX interrupt are enabled when you call this API.

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	ENET handler pointer.
<i>ptpConfig</i>	The ENET PTP1588 configuration.

13.7.33 void ENET_Ptp1588StartTimer (ENET_Type * *base*, uint32_t *ptpClkSrc*)

This function is used to initialize the PTP timer. After the PTP starts, the PTP timer starts running.

Function Documentation

Parameters

<i>base</i>	ENET peripheral base address.
<i>ptpClkSrc</i>	The clock source of the PTP timer.

13.7.34 static void ENET_Ptp1588StopTimer (ENET_Type * *base*) [inline], [static]

This function is used to stops the ENET PTP timer.

Parameters

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

13.7.35 void ENET_Ptp1588AdjustTimer (ENET_Type * *base*, uint32_t *corrIncrease*, uint32_t *corrPeriod*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>corrIncrease</i>	The correction increment value. This value is added every time the correction timer expires. A value less than the PTP timer frequency(1/ptpClkSrc) slows down the timer, a value greater than the 1/ptpClkSrc speeds up the timer.
<i>corrPeriod</i>	The PTP timer correction counter wrap-around value. This defines after how many timer clock the correction counter should be reset and trigger a correction increment on the timer. A value of 0 disables the correction counter and no correction occurs.

13.7.36 static void ENET_Ptp1588SetChannelMode (ENET_Type * *base*, enet_ptp_timer_channel_t *channel*, enet_ptp_timer_channel_mode_t *mode*, bool *intEnable*) [inline], [static]

Parameters

<i>base</i>	ENET peripheral base address.
<i>channel</i>	The ENET PTP timer channel number.
<i>mode</i>	The PTP timer channel mode, see "enet_ptp_timer_channel_mode_t".
<i>intEnable</i>	Enables or disables the interrupt.

13.7.37 static void ENET_Ptp1588SetChannelCmpValue (ENET_Type * *base*, enet_ptp_timer_channel_t *channel*, uint32_t *cmpValue*) [inline], [static]

Parameters

<i>base</i>	ENET peripheral base address.
<i>channel</i>	The PTP timer channel, see "enet_ptp_timer_channel_t".
<i>cmpValue</i>	The compare value for the compare setting.

13.7.38 static bool ENET_Ptp1588GetChannelStatus (ENET_Type * *base*, enet_ptp_timer_channel_t *channel*) [inline], [static]

Parameters

<i>base</i>	ENET peripheral base address.
<i>channel</i>	The IEEE 1588 timer channel number.

Returns

True or false, Compare or capture operation status

13.7.39 static void ENET_Ptp1588ClearChannelStatus (ENET_Type * *base*, enet_ptp_timer_channel_t *channel*) [inline], [static]

Parameters

Function Documentation

<i>base</i>	ENET peripheral base address.
<i>channel</i>	The IEEE 1588 timer channel number.

13.7.40 void ENET_Ptp1588GetTimer (ENET_Type * *base*, enet_handle_t * *handle*, enet_ptp_time_t * *ptpTime*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET state pointer. This is the same state pointer used in the ENET_Init.
<i>ptpTime</i>	The PTP timer structure.

13.7.41 void ENET_Ptp1588SetTimer (ENET_Type * *base*, enet_handle_t * *handle*, enet_ptp_time_t * *ptpTime*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET state pointer. This is the same state pointer used in the ENET_Init.
<i>ptpTime</i>	The timer to be set to the PTP timer.

13.7.42 void ENET_Ptp1588TimerIRQHandler (ENET_Type * *base*, enet_handle_t * *handle*)

Parameters

<i>base</i>	ENET peripheral base address.
<i>handle</i>	The ENET state pointer. This is the same state pointer used in the ENET_Init.

13.7.43 status_t ENET_GetRxFrameTime (enet_handle_t * *handle*, enet_ptp_time_data_t * *ptpTimeData*)

This function is used for PTP stack to get the timestamp captured by the ENET driver.

Parameters

<i>handle</i>	The ENET handler pointer. This is the same state pointer used in ENET_Init.
<i>ptpTimeData</i>	The special PTP timestamp data for search the receive timestamp.

Return values

<i>kStatus_Success</i>	Get 1588 timestamp success.
<i>kStatus_ENET_PtpTs-RingEmpty</i>	1588 timestamp ring empty.
<i>kStatus_ENET_PtpTs-RingFull</i>	1588 timestamp ring full.

13.7.44 status_t ENET_GetTxFrameTime (enet_handle_t * *handle*, enet_ptp_time_data_t * *ptpTimeData*)

This function is used for PTP stack to get the timestamp captured by the ENET driver.

Parameters

<i>handle</i>	The ENET handler pointer. This is the same state pointer used in ENET_Init.
<i>ptpTimeData</i>	The special PTP timestamp data for search the receive timestamp.

Return values

<i>kStatus_Success</i>	Get 1588 timestamp success.
<i>kStatus_ENET_PtpTs-RingEmpty</i>	1588 timestamp ring empty.
<i>kStatus_ENET_PtpTs-RingFull</i>	1588 timestamp ring full.

Function Documentation

Chapter 14

EWM: External Watchdog Monitor Driver

14.1 Overview

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

14.2 Typical use case

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

Data Structures

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

Enumerations

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

Driver version

- #define `FSL_EWM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 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)
Disables the EWM interrupt.
- static `uint32_t` `EWM_GetStatusFlags` (`EWM_Type` *base)
Gets all status flags.
- void `EWM_Refresh` (`EWM_Type` *base)
Services the EWM.

Enumeration Type Documentation

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 enables the EWM interrupt.

Parameters

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

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

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.

Chapter 16

FlexBus: External Bus Interface Driver

16.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Crossbar External Bus Interface (FlexBus) block of MCUXpresso SDK 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.

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

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,

Typical use case and example

- ```
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, 2))  
*Version 2.0.2.*

## 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**  
*Auto acknowledge setting.*
- bool **extendTransferAddress**  
*Extend transfer start/extend address latch enable.*
- bool **secondaryWaitStates**  
*Secondary wait states number.*
- flexbus\_port\_size\_t **portSize**  
*Port size of transfer.*
- flexbus\_bytelane\_shift\_t **byteLaneShift**  
*Byte-lane shift enable.*
- flexbus\_write\_address\_hold\_t **writeAddressHold**

## Enumeration Type Documentation

- *Write address hold or deselect option.*
  - **flexbus\_read\_address\_hold\_t** `readAddressHold`  
*Read address hold or deselect option.*
  - **flexbus\_address\_setup\_t** `addressSetup`  
*Address setup setting.*
- **flexbus\_multiplex\_group1\_t** `group1MultiplexControl`  
*FlexBus Signal Group 1 Multiplex control.*
- **flexbus\_multiplex\_group2\_t** `group2MultiplexControl`  
*FlexBus Signal Group 2 Multiplex control.*
- **flexbus\_multiplex\_group3\_t** `group3MultiplexControl`  
*FlexBus Signal Group 3 Multiplex control.*
- **flexbus\_multiplex\_group4\_t** `group4MultiplexControl`  
*FlexBus Signal Group 4 Multiplex control.*
- **flexbus\_multiplex\_group5\_t** `group5MultiplexControl`  
*FlexBus Signal Group 5 Multiplex control.*

## 16.5 Macro Definition Documentation

### 16.5.1 `#define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))`

## 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.

#### 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.

## Function Documentation

### 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;

```

## Parameters

|               |                                          |
|---------------|------------------------------------------|
| <i>config</i> | Pointer to the initialization structure. |
|---------------|------------------------------------------|

## See Also

[FLEXBUS\\_Init](#)

## Function Documentation

# Chapter 17

## FlexCAN: Flex Controller Area Network Driver

### 17.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Flex Controller Area Network (FlexCAN) module of MCUXpresso SDK 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

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

##### 17.2.2.2 Message Buffer Receive Operation

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

##### 17.2.2.3 Receive FIFO Operation

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

## Data Structures

- struct [flexcan\\_frame\\_t](#)  
*FlexCAN message frame structure. [More...](#)*
- struct [flexcan\\_timing\\_config\\_t](#)  
*FlexCAN protocol timing characteristic configuration structure. [More...](#)*
- struct [flexcan\\_config\\_t](#)  
*FlexCAN module configuration structure. [More...](#)*
- struct [flexcan\\_rx\\_mb\\_config\\_t](#)  
*FlexCAN Receive Message Buffer configuration structure. [More...](#)*
- struct [flexcan\\_rx\\_fifo\\_config\\_t](#)  
*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) (((uint32\_t)(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) (((uint32\_t)(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) (((uint32\_t)(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) (((uint32\_t)(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)

## FlexCAN Driver

- 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)
- 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)
  - 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, 9),
   
  `kStatus_FLEXCAN_ErrorStatus` = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 10),
   
  `kStatus_FLEXCAN_WakeUp` = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 11),
   
  `kStatus_FLEXCAN_UnHandled` = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 12) }
- 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_wake_up_source_t` {
   
  `kFLEXCAN_WakeupSrcUnfiltered` = 0x0U,
   
  `kFLEXCAN_WakeupSrcFiltered` = 0x1U }
- FlexCAN wake up 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` {

## FlexCAN Driver

```
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 }

*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 }
- 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 `FSL_FLEXCAN_DRIVER_VERSION` (MAKE\_VERSION(2, 3, 2))  
*FlexCAN driver version 2.3.0.*

## Initialization and deinitialization

- uint32\_t `FLEXCAN_GetInstance` (CAN\_Type \*base)  
*Get the FlexCAN instance from peripheral base address.*
- 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.*

## 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.*

## FlexCAN Driver

### 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

#### 17.2.3.1.0.40 Field Documentation

17.2.3.1.0.40.1 uint32\_t flexcan\_frame\_t::timestamp

17.2.3.1.0.40.2 uint32\_t flexcan\_frame\_t::length

17.2.3.1.0.40.3 uint32\_t flexcan\_frame\_t::type

17.2.3.1.0.40.4 uint32\_t flexcan\_frame\_t::format

17.2.3.1.0.40.5 uint32\_t flexcan\_frame\_t::\_\_pad0\_\_

17.2.3.1.0.40.6 uint32\_t flexcan\_frame\_t::idhit

17.2.3.1.0.40.7 uint32\_t flexcan\_frame\_t::id

17.2.3.1.0.40.8 uint32\_t flexcan\_frame\_t::dataWord0

17.2.3.1.0.40.9 uint32\_t flexcan\_frame\_t::dataWord1

17.2.3.1.0.40.10 uint8\_t flexcan\_frame\_t::dataByte3

17.2.3.1.0.40.11 uint8\_t flexcan\_frame\_t::dataByte2

17.2.3.1.0.40.12 uint8\_t flexcan\_frame\_t::dataByte1

17.2.3.1.0.40.13 uint8\_t flexcan\_frame\_t::dataByte0

17.2.3.1.0.40.14 uint8\_t flexcan\_frame\_t::dataByte7

17.2.3.1.0.40.15 uint8\_t flexcan\_frame\_t::dataByte6

17.2.3.1.0.40.16 uint8\_t flexcan\_frame\_t::dataByte5

17.2.3.1.0.40.17 uint8\_t flexcan\_frame\_t::dataByte4

### 17.2.3.2 struct flexcan\_timing\_config\_t

#### Data Fields

- uint16\_t preDivider  
*Clock Pre-scaler Division Factor.*
- uint8\_t rJumpwidth  
*Re-sync Jump Width.*
- uint8\_t phaseSeg1  
*Phase Segment 1.*

## FlexCAN Driver

- `uint8_t phaseSeg2`  
*Phase Segment 2.*
- `uint8_t propSeg`  
*Propagation Segment.*

### 17.2.3.2.0.41 Field Documentation

17.2.3.2.0.41.1 `uint16_t flexcan_timing_config_t::preDivider`

17.2.3.2.0.41.2 `uint8_t flexcan_timing_config_t::rJumpwidth`

17.2.3.2.0.41.3 `uint8_t flexcan_timing_config_t::phaseSeg1`

17.2.3.2.0.41.4 `uint8_t flexcan_timing_config_t::phaseSeg2`

17.2.3.2.0.41.5 `uint8_t flexcan_timing_config_t::propSeg`

### 17.2.3.3 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.*
- `flexcan_wake_up_source_t wakeupSrc`  
*Wake up source selection.*
- `uint8_t maxMbNum`  
*The maximum number of Message Buffers used by user.*
- `bool enableLoopBack`  
*Enable or Disable Loop Back Self Test Mode.*
- `bool enableTimerSync`  
*Enable or Disable Timer Synchronization.*
- `bool enableSelfWakeup`  
*Enable or Disable Self Wakeup Mode.*
- `bool enableIndividMask`  
*Enable or Disable Rx Individual Mask.*

### 17.2.3.3.0.42 Field Documentation

17.2.3.3.0.42.1 `uint32_t flexcan_config_t::baudRate`

17.2.3.3.0.42.2 `flexcan_clock_source_t flexcan_config_t::clkSrc`

17.2.3.3.0.42.3 `flexcan_wake_up_source_t flexcan_config_t::wakeupSrc`

17.2.3.3.0.42.4 `uint8_t flexcan_config_t::maxMbNum`

17.2.3.3.0.42.5 `bool flexcan_config_t::enableLoopBack`

17.2.3.3.0.42.6 `bool flexcan_config_t::enableTimerSync`

17.2.3.3.0.42.7 `bool flexcan_config_t::enableSelfWakeup`

17.2.3.3.0.42.8 `bool flexcan_config_t::enableIndividMask`

### 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).*

### 17.2.3.4.0.43 Field Documentation

17.2.3.4.0.43.1 `uint32_t flexcan_rx_mb_config_t::id`

17.2.3.4.0.43.2 `flexcan_frame_format_t flexcan_rx_mb_config_t::format`

17.2.3.4.0.43.3 `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.*

## FlexCAN Driver

- `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.*

### 17.2.3.5.0.44 Field Documentation

17.2.3.5.0.44.1 `uint32_t* flexcan_rx_fifo_config_t::idFilterTable`

17.2.3.5.0.44.2 `uint8_t flexcan_rx_fifo_config_t::idFilterNum`

17.2.3.5.0.44.3 `flexcan_rx_fifo_filter_type_t flexcan_rx_fifo_config_t::idFilterType`

17.2.3.5.0.44.4 `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.*

### 17.2.3.6.0.45 Field Documentation

17.2.3.6.0.45.1 `flexcan_frame_t* flexcan_mb_transfer_t::frame`

17.2.3.6.0.45.2 `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.*

### 17.2.3.7.0.46 Field Documentation

17.2.3.7.0.46.1 `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.*
- `volatile uint8_t mbState [CAN_WORD1_COUNT]`  
*Message Buffer transfer state.*
- `volatile uint8_t rxFifoState`  
*Rx FIFO transfer state.*

#### 17.2.3.8.0.47 Field Documentation

**17.2.3.8.0.47.1 `flexcan_transfer_callback_t flexcan_handle_t::callback`**

**17.2.3.8.0.47.2 `void* flexcan_handle_t::userData`**

**17.2.3.8.0.47.3 `flexcan_frame_t* volatile flexcan_handle_t::mbFrameBuf[CAN_WORD1_COUNT]`**

**17.2.3.8.0.47.4 `flexcan_frame_t* volatile flexcan_handle_t::rxFifoFrameBuf`**

**17.2.3.8.0.47.5 `volatile uint8_t flexcan_handle_t::mbState[CAN_WORD1_COUNT]`**

**17.2.3.8.0.47.6 `volatile uint8_t flexcan_handle_t::rxFifoState`**

#### 17.2.4 Macro Definition Documentation

**17.2.4.1 `#define FSL_FLEXCAN_DRIVER_VERSION (MAKE_VERSION(2, 3, 2))`**

**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))
```

## FlexCAN Driver

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)) | \
(((uint32_t)(id)&0x7FF) << 19))
```

### 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)) | \
(((uint32_t)(id)&0x7FF) << 3))
```

- 17.2.4.9 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_HIGH( *id*  
   ) (((uint32\_t)(*id*)&0x7F8) << 21)
- 17.2.4.10 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_MID\_HIGH( *id*  
   ) (((uint32\_t)(*id*)&0x7F8) << 13)
- 17.2.4.11 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_MID\_LOW( *id*  
   ) (((uint32\_t)(*id*)&0x7F8) << 5)
- 17.2.4.12 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_LOW( *id*  
   ) (((uint32\_t)(*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) \
```

## FlexCAN Driver

**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 ) ((FLEXCAN\_ID\_EXT(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)
```

**17.2.4.26 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_LOW( *id* )****Value:**

```
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* )****Value:**

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(\
 id)
```

## FlexCAN Driver

### 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\_WakeUp* FlexCAN is waken up from STOP mode.  
*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.

### 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\_wake\_up\_source\_t

Enumerator

*kFLEXCAN\_WakeupSrcUnfiltered* FlexCAN uses unfiltered Rx input to detect edge.  
*kFLEXCAN\_WakeupSrcFiltered* FlexCAN uses filtered Rx input to detect edge.

### 17.2.6.6 enum flexcan\_rx\_fifo\_filter\_type\_t

Enumerator

*kFLEXCAN\_RxFifoFilterTypeA* One full ID (standard and extended) per ID Filter element.

## FlexCAN Driver

***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.

### 17.2.6.7 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.

### 17.2.6.8 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.

### 17.2.6.9 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.

### 17.2.6.10 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.

### 17.2.6.11 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 uint32\_t FLEXCAN\_GetInstance ( CAN\_Type \* base )

## FlexCAN Driver

Parameters

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

Returns

FlexCAN instance.

### 17.2.7.2 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.

```
* flexcan_config_t flexcanConfig;
* flexcanConfig.clkSrc = kFLEXCAN_ClkSrcOsc;
* flexcanConfig.baudRate = 1000000U;
* flexcanConfig.maxMbNum = 16;
* flexcanConfig.enableLoopBack = false;
* flexcanConfig.enableSelfWakeup = false;
* flexcanConfig.enableIndividMask = false;
* flexcanConfig.enableDoze = false;
* flexcanConfig.timingConfig = timingConfig;
* 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. |

### 17.2.7.3 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. |
|-------------|----------------------------------|

#### 17.2.7.4 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 = 1000000U;` `flexcanConfig->baudRateFD = 2000000U;` `flexcanConfig->maxMbNum = 16;` `flexcanConfig->enableLoopBack = false;` `flexcanConfig->enableSelfWakeup = false;` `flexcanConfig->enableIndividMask = false;` `flexcanConfig->enableDoze = false;` `flexcanConfig.timingConfig = timingConfig;`

Parameters

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

#### 17.2.7.5 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. |

#### 17.2.7.6 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. |
|-------------|--------------------------------------|

### 17.2.7.7 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.       |

### 17.2.7.8 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.        |

### 17.2.7.9 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"> <li>• true: Enable Tx Message Buffer.</li> <li>• false: Disable Tx Message Buffer.</li> </ul> |

#### 17.2.7.10 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"> <li>• true: Enable Rx Message Buffer.</li> <li>• false: Disable Rx Message Buffer.</li> </ul> |

#### 17.2.7.11 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"> <li>• true: Enable Rx FIFO.</li> <li>• false: Disable Rx FIFO.</li> </ul> |

#### 17.2.7.12 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`.

#### 17.2.7.13 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> . |

#### 17.2.7.14 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. |

#### 17.2.7.15 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.

### 17.2.7.16 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. |

### 17.2.7.17 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 <a href="#">_flexcan_interrupt_enable</a> . |

### 17.2.7.18 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 <a href="#">_flexcan_interrupt_enable</a> . |

#### 17.2.7.19 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. |

#### 17.2.7.20 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. |

#### 17.2.7.21 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. |

#### 17.2.7.22 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. |

### 17.2.7.23 **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_Rx-Overflow</i> | - Rx Message Buffer is already overflowed and has been read successfully. |
| <i>kStatus_Fail</i>                | - Rx Message Buffer is empty.                                             |

### 17.2.7.24 **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.                 |

#### 17.2.7.25 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. |

#### 17.2.7.26 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.                                             |

### 17.2.7.27 **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.                 |

### 17.2.7.28 **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. |

### 17.2.7.29 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 <a href="#">flexcan_mb_transfer_t</a> . |

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.                          |

### 17.2.7.30 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 <a href="#">flexcan_mb_transfer_t</a> . |

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.                            |

### 17.2.7.31 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 <a href="#">flexcan_fifo_transfer_t</a> . |

Return values

|                                    |                                                 |
|------------------------------------|-------------------------------------------------|
| <i>kStatus_Success</i>             | - Start Rx FIFO receiving process successfully. |
| <i>kStatus_FLEXCAN_Rx-FifoBusy</i> | - Rx FIFO is currently in use.                  |

### 17.2.7.32 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. |

### 17.2.7.33 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. |

### 17.2.7.34 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.          |

### 17.2.7.35 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.*

#### Driver version

- #define `FSL_FLEXCAN_EDMA_DRIVER_VERSION` (MAKE\_VERSION(2, 2, 4))  
*FlexCAN EDMA driver version 2.2.4.*

#### 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`

## FlexCAN eDMA Driver

- volatile uint8\_t **rxFifoState**  
*Rx FIFO transfer state.*

### 17.3.2.1.0.48 Field Documentation

17.3.2.1.0.48.1 **flexcan\_edma\_transfer\_callback\_t flexcan\_edma\_handle\_t::callback**

17.3.2.1.0.48.2 **void\* flexcan\_edma\_handle\_t::userData**

17.3.2.1.0.48.3 **edma\_handle\_t\* flexcan\_edma\_handle\_t::rxFifoEdmaHandle**

17.3.2.1.0.48.4 **volatile uint8\_t flexcan\_edma\_handle\_t::rxFifoState**

### 17.3.3 Macro Definition Documentation

17.3.3.1 **#define FSL\_FLEXCAN\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 4))**

### 17.3.4 Typedef Documentation

17.3.4.1 **typedef void(\* flexcan\_edma\_transfer\_callback\_t)(CAN\_Type \*base, flexcan\_edma\_handle\_t \*handle, status\_t status, void \*userData)**

### 17.3.5 Function Documentation

17.3.5.1 **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. |

17.3.5.2 **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 <a href="#">flexcan_fifo_transfer_t</a> . |

Return values

|                                    |                            |
|------------------------------------|----------------------------|
| <i>kStatus_Success</i>             | if succeed, others failed. |
| <i>kStatus_FLEXCAN_Rx-FifoBusy</i> | Previous transfer ongoing. |

### 17.3.5.3 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.            |
| <i>handle</i> | Pointer to flexcan_edma_handle_t structure. |



# Chapter 18

## FTM: FlexTimer Driver

### 18.1 Overview

The MCUXpresso SDK provides a driver for the FlexTimer Module (FTM) of MCUXpresso SDK 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. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ftmMultiple PWM synchronization update modes can be used by providing an OR'ed list of options available in the enumeration [ftm\\_pwm\\_sync\\_method\\_t](#) to the pwmSyncMode field.

When using an intermediate reload points, the PWM synchronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in the enumeration [ftm\\_reload\\_point\\_t](#) to the reloadPoints field.

The driver initialization function sets up the appropriate bits in the FTM module based on the register update options selected.

If software PWM synchronization is used, the below function can be used to initiate a software trigger. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ftm

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

## Data Structures

- struct `ftm_chnl_pwm_signal_param_t`  
*Options to configure a FTM channel's PWM signal.* [More...](#)
- struct `ftm_chnl_pwm_config_param_t`  
*Options to configure a FTM channel using precise setting.* [More...](#)
- struct `ftm_dual_edge_capture_param_t`  
*FlexTimer dual edge capture parameters.* [More...](#)
- struct `ftm_phase_params_t`  
*FlexTimer quadrature decode phase parameters.* [More...](#)
- struct `ftm_fault_param_t`  
*Structure is used to hold the parameters to configure a FTM fault.* [More...](#)
- struct `ftm_config_t`  
*FTM configuration structure.* [More...](#)

## Enumerations

- enum `ftm_chnl_t` {
   
kFTM\_Chnl\_0 = 0U,
   
kFTM\_Chnl\_1,
   
kFTM\_Chnl\_2,
   
kFTM\_Chnl\_3,
   
kFTM\_Chnl\_4,
   
kFTM\_Chnl\_5,
   
kFTM\_Chnl\_6,
   
kFTM\_Chnl\_7 }
   
*List of FTM channels.*
- enum `ftm_fault_input_t` {
   
kFTM\_Fault\_0 = 0U,
   
kFTM\_Fault\_1,
   
kFTM\_Fault\_2,
   
kFTM\_Fault\_3 }
   
*List of FTM faults.*
- enum `ftm_pwm_mode_t` {
   
kFTM\_EdgeAlignedPwm = 0U,
   
kFTM\_CenterAlignedPwm,
   
kFTM\_CombinedPwm }
   
*FTM PWM operation modes.*
- enum `ftm_pwm_level_select_t` {

## Typical use case

```
kFTM_NoPwmSignal = 0U,
kFTM_LowTrue,
kFTM_HighTrue }
```

*FTM PWM output pulse mode: high-true, low-true or no output.*

- enum `ftm_output_compare_mode_t` {  
  `kFTM_NoOutputSignal` = (1U << FTM\_CnSC\_MSA\_SHIFT),  
  `kFTM_ToggleOnMatch` = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (1U << FTM\_CnSC\_ELSA\_S-HIFT)),  
  `kFTM_ClearOnMatch` = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (2U << FTM\_CnSC\_ELSA\_SHIFT)),  
  `kFTM_SetOnMatch` = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (3U << FTM\_CnSC\_ELSA\_SHIFT)) }

*FlexTimer output compare mode.*

- enum `ftm_input_capture_edge_t` {  
  `kFTM_RisingEdge` = (1U << FTM\_CnSC\_ELSA\_SHIFT),  
  `kFTM_FallingEdge` = (2U << FTM\_CnSC\_ELSA\_SHIFT),  
  `kFTM_RiseAndFallEdge` = (3U << FTM\_CnSC\_ELSA\_SHIFT) }

*FlexTimer input capture edge.*

- enum `ftm_dual_edge_capture_mode_t` {  
  `kFTM_OneShot` = 0U,  
  `kFTM_Continuous` = (1U << FTM\_CnSC\_MSA\_SHIFT) }

*FlexTimer dual edge capture modes.*

- enum `ftm_quad_decode_mode_t` {  
  `kFTM_QuadPhaseEncode` = 0U,  
  `kFTM_QuadCountAndDir` }
- enum `ftm_phase_polarity_t` {  
  `kFTM_QuadPhaseNormal` = 0U,  
  `kFTM_QuadPhaseInvert` }

*FlexTimer quadrature phase polarities.*

- enum `ftm_deadtime_prescale_t` {  
  `kFTM_Deadtime_Prescale_1` = 1U,  
  `kFTM_Deadtime_Prescale_4`,  
  `kFTM_Deadtime_Prescale_16` }

*FlexTimer pre-scaler factor for the dead time insertion.*

- enum `ftm_clock_source_t` {  
  `kFTM_SystemClock` = 1U,  
  `kFTM_FixedClock`,  
  `kFTM_ExternalClock` }

*FlexTimer clock source selection.*

- enum `ftm_clock_prescale_t` {

```
kFTM_Prescale_Divide_1 = 0U,
kFTM_Prescale_Divide_2,
kFTM_Prescale_Divide_4,
kFTM_Prescale_Divide_8,
kFTM_Prescale_Divide_16,
kFTM_Prescale_Divide_32,
kFTM_Prescale_Divide_64,
kFTM_Prescale_Divide_128 }
```

*FlexTimer pre-scaler factor selection for the clock source.*

- enum `ftm_bdm_mode_t` {
   
kFTM\_BdmMode\_0 = 0U,
   
kFTM\_BdmMode\_1,
   
kFTM\_BdmMode\_2,
   
kFTM\_BdmMode\_3 }

*Options for the FlexTimer behaviour in BDM Mode.*

- enum `ftm_fault_mode_t` {
   
kFTM\_Fault\_Disable = 0U,
   
kFTM\_Fault\_EvenChnls,
   
kFTM\_Fault\_AllChnlsMan,
   
kFTM\_Fault\_AllChnlsAuto }

*Options for the FTM fault control mode.*

- enum `ftm_external_trigger_t` {
   
kFTM\_Chnl0Trigger = (1U << 4),
   
kFTM\_Chnl1Trigger = (1U << 5),
   
kFTM\_Chnl2Trigger = (1U << 0),
   
kFTM\_Chnl3Trigger = (1U << 1),
   
kFTM\_Chnl4Trigger = (1U << 2),
   
kFTM\_Chnl5Trigger = (1U << 3),
   
kFTM\_InitTrigger = (1U << 6) }

*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` {

## Typical use case

```
kFTM_Chnl0Match = (1U << 0),
kFTM_Chnl1Match = (1U << 1),
kFTM_Chnl2Match = (1U << 2),
kFTM_Chnl3Match = (1U << 3),
kFTM_Chnl4Match = (1U << 4),
kFTM_Chnl5Match = (1U << 5),
kFTM_Chnl6Match = (1U << 6),
kFTM_Chnl7Match = (1U << 7),
kFTM_CntMax = (1U << 8),
kFTM_CntMin = (1U << 9),
kFTM_HalfCycMatch = (1U << 10) }
```

*FTM options available as loading point for register reload.*

- enum `ftm_interrupt_enable_t` {  
    kFTM\_Chnl0InterruptEnable = (1U << 0),  
    kFTM\_Chnl1InterruptEnable = (1U << 1),  
    kFTM\_Chnl2InterruptEnable = (1U << 2),  
    kFTM\_Chnl3InterruptEnable = (1U << 3),  
    kFTM\_Chnl4InterruptEnable = (1U << 4),  
    kFTM\_Chnl5InterruptEnable = (1U << 5),  
    kFTM\_Chnl6InterruptEnable = (1U << 6),  
    kFTM\_Chnl7InterruptEnable = (1U << 7),  
    kFTM\_FaultInterruptEnable = (1U << 8),  
    kFTM\_TimeOverflowInterruptEnable = (1U << 9),  
    kFTM\_ReloadInterruptEnable = (1U << 10) }

*List of FTM interrupts.*

- enum `ftm_status_flags_t` {  
    kFTM\_Chnl0Flag = (1U << 0),  
    kFTM\_Chnl1Flag = (1U << 1),  
    kFTM\_Chnl2Flag = (1U << 2),  
    kFTM\_Chnl3Flag = (1U << 3),  
    kFTM\_Chnl4Flag = (1U << 4),  
    kFTM\_Chnl5Flag = (1U << 5),  
    kFTM\_Chnl6Flag = (1U << 6),  
    kFTM\_Chnl7Flag = (1U << 7),  
    kFTM\_FaultFlag = (1U << 8),  
    kFTM\_TimeOverflowFlag = (1U << 9),  
    kFTM\_ChnlTriggerFlag = (1U << 10),  
    kFTM\_ReloadFlag = (1U << 11) }

*List of FTM flags.*

- enum `_ftm_quad_decoder_flags` {  
    kFTM\_QuadDecoderCountingIncreaseFlag = FTM\_QDCTRL\_QUADIR\_MASK,  
    kFTM\_QuadDecoderCountingOverflowOnTopFlag = FTM\_QDCTRL\_TOFDIR\_MASK }

*List of FTM Quad Decoder flags.*

## Functions

- void **FTM\_SetupFault** (FTM\_Type \*base, **ftm\_fault\_input\_t** faultNumber, const **ftm\_fault\_param\_t** \*faultParams)
 

*Sets up the working of the FTM fault protection.*
- static void **FTM\_SetGlobalTimeBaseOutputEnable** (FTM\_Type \*base, bool enable)
 

*Enables or disables the FTM global time base signal generation to other FTMs.*
- static void **FTM\_SetOutputMask** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool mask)
 

*Sets the FTM peripheral timer channel output mask.*
- static void **FTM\_SetSoftwareTrigger** (FTM\_Type \*base, bool enable)
 

*Enables or disables the FTM software trigger for PWM synchronization.*
- static void **FTM\_SetWriteProtection** (FTM\_Type \*base, bool enable)
 

*Enables or disables the FTM write protection.*

## Driver version

- #define **FSL\_FTM\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 0))
 

*FTM driver version 2.1.0.*

## Initialization and deinitialization

- status\_t **FTM\_Init** (FTM\_Type \*base, const **ftm\_config\_t** \*config)
 

*Ungates the FTM clock and configures the peripheral for basic operation.*
- void **FTM\_Deinit** (FTM\_Type \*base)
 

*Gates the FTM clock.*
- void **FTM\_GetDefaultConfig** (**ftm\_config\_t** \*config)
 

*Fills in the FTM configuration structure with the default settings.*

## 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.*
- status\_t **FTM\_SetupPwmMode** (FTM\_Type \*base, const **ftm\_chnl\_pwm\_config\_param\_t** \*chnlParams, uint8\_t numOfChnls, **ftm\_pwm\_mode\_t** mode)
 

*Configures the PWM mode parameters.*
- void **FTM\_SetupInputCapture** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_input\_capture\_edge\_t** captureMode, uint32\_t filterValue)
 

*Enables capturing an input signal on the channel using the function parameters.*
- void **FTM\_SetupOutputCompare** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_output\_compare\_mode\_t** compareMode, uint32\_t compareValue)
 

*Configures the FTM to generate timed pulses.*
- void **FTM\_SetupDualEdgeCapture** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, const **ftm\_dual\_edge\_capture\_param\_t** \*edgeParam, uint32\_t filterValue)
 

*Configures the dual edge capture mode of the FTM.*

## Typical use case

### Interrupt Interface

- void **FTM\_EnableInterrupts** (FTM\_Type \*base, uint32\_t mask)  
*Enables the selected FTM interrupts.*
- void **FTM\_DisableInterrupts** (FTM\_Type \*base, uint32\_t mask)  
*Disables the selected FTM interrupts.*
- uint32\_t **FTM\_GetEnabledInterrupts** (FTM\_Type \*base)  
*Gets the enabled FTM interrupts.*

### Status Interface

- uint32\_t **FTM\_GetStatusFlags** (FTM\_Type \*base)  
*Gets the FTM status flags.*
- void **FTM\_ClearStatusFlags** (FTM\_Type \*base, uint32\_t mask)  
*Clears the FTM status flags.*

### Read and write the timer period

- static void **FTM\_SetTimerPeriod** (FTM\_Type \*base, uint32\_t ticks)  
*Sets the timer period in units of ticks.*
- static uint32\_t **FTM\_GetCurrentTimerCount** (FTM\_Type \*base)  
*Reads the current timer counting value.*

### Timer Start and Stop

- static void **FTM\_StartTimer** (FTM\_Type \*base, **ftm\_clock\_source\_t** clockSource)  
*Starts the FTM counter.*
- static void **FTM\_StopTimer** (FTM\_Type \*base)  
*Stops the FTM counter.*

### Software output control

- static void **FTM\_SetSoftwareCtrlEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool value)  
*Enables or disables the channel software output control.*
- static void **FTM\_SetSoftwareCtrlVal** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool value)  
*Sets the channel software output control value.*

### Channel pair operations

- static void **FTM\_SetFaultControlEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
*This function enables/disables the fault control in a channel pair.*
- static void **FTM\_SetDeadTimeEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
*This function enables/disables the dead time insertion in a channel pair.*
- static void **FTM\_SetComplementaryEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
*This function enables/disables complementary mode in a channel pair.*
- static void **FTM\_SetInvertEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
*This function enables/disables inverting control in a channel pair.*

## Quad Decoder

- void [FTM\\_SetupQuadDecode](#) (FTM\_Type \*base, const [ftm\\_phase\\_params\\_t](#) \*phaseAParams, const [ftm\\_phase\\_params\\_t](#) \*phaseBParams, [ftm\\_quad\\_decode\\_mode\\_t](#) quadMode)  
*Configures the parameters and activates the quadrature decoder mode.*
- static uint32\_t [FTM\\_GetQuadDecoderFlags](#) (FTM\_Type \*base)  
*Gets the FTM Quad Decoder flags.*
- static void [FTM\\_SetQuadDecoderModuloValue](#) (FTM\_Type \*base, uint32\_t startValue, uint32\_t overValue)  
*Sets the modulo values for Quad Decoder.*
- static uint32\_t [FTM\\_GetQuadDecoderCounterValue](#) (FTM\_Type \*base)  
*Gets the current Quad Decoder counter value.*
- static void [FTM\\_ClearQuadDecoderCounterValue](#) (FTM\_Type \*base)  
*Clears the current Quad Decoder counter value.*

## 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.*
- 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.*

#### 18.5.1.0.0.49 Field Documentation

##### 18.5.1.0.0.49.1 [ftm\\_chnl\\_t ftm\\_chnl\\_pwm\\_signal\\_param\\_t::chnlNumber](#)

In combined mode, this represents the channel pair number.

##### 18.5.1.0.0.49.2 [ftm\\_pwm\\_level\\_select\\_t ftm\\_chnl\\_pwm\\_signal\\_param\\_t::level](#)

##### 18.5.1.0.0.49.3 [uint8\\_t ftm\\_chnl\\_pwm\\_signal\\_param\\_t::dutyCyclePercent](#)

100 = always active signal (100% duty cycle).

##### 18.5.1.0.0.49.4 [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

## Data Structure Documentation

### 18.5.2 struct ftm\_chnl\_pwm\_config\_param\_t

#### Data Fields

- **ftm\_chnl\_t chnlNumber**  
*The channel/channel pair number.*
- **ftm\_pwm\_level\_select\_t level**  
*PWM output active level select.*
- **uint16\_t dutyValue**  
*PWM pulse width, the uint of this value is timer ticks.*
- **uint16\_t firstEdgeValue**  
*Used only in combined PWM mode to generate an asymmetrical PWM.*

#### 18.5.2.0.0.50 Field Documentation

##### 18.5.2.0.0.50.1 ftm\_chnl\_t ftm\_chnl\_pwm\_config\_param\_t::chnlNumber

In combined mode, this represents the channel pair number.

##### 18.5.2.0.0.50.2 ftm\_pwm\_level\_select\_t ftm\_chnl\_pwm\_config\_param\_t::level

##### 18.5.2.0.0.50.3 uint16\_t ftm\_chnl\_pwm\_config\_param\_t::dutyValue

##### 18.5.2.0.0.50.4 uint16\_t ftm\_chnl\_pwm\_config\_param\_t::firstEdgeValue

Specifies the delay to the first edge in a PWM period. If unsure leave as 0, uint of this value is timer ticks.

### 18.5.3 struct ftm\_dual\_edge\_capture\_param\_t

#### Data Fields

- **ftm\_dual\_edge\_capture\_mode\_t mode**  
*Dual Edge Capture mode.*
- **ftm\_input\_capture\_edge\_t currChanEdgeMode**  
*Input capture edge select for channel n.*
- **ftm\_input\_capture\_edge\_t nextChanEdgeMode**  
*Input capture edge select for channel n+1.*

### 18.5.4 struct ftm\_phase\_params\_t

#### Data Fields

- **bool enablePhaseFilter**  
*True: enable phase filter; false: disable filter.*
- **uint32\_t phaseFilterVal**

- **ftm\_phase\_polarity\_t phasePolarity**  
*Phase polarity.*
- Filter value, used only if phase filter is enabled.*

### 18.5.5 struct ftm\_fault\_param\_t

#### Data Fields

- bool **enableFaultInput**  
*True: Fault input is enabled; false: Fault input is disabled.*
- bool **faultLevel**  
*True: Fault polarity is active low; in other words, '0' indicates a fault; False: Fault polarity is active high.*
- bool **useFaultFilter**  
*True: Use the filtered fault signal; False: Use the direct path from fault input.*

### 18.5.6 struct ftm\_config\_t

This structure holds the configuration settings for the FTM peripheral. To initialize this structure to reasonable defaults, call the [FTM\\_GetDefaultConfig\(\)](#) function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

#### Data Fields

- **ftm\_clock\_prescale\_t prescale**  
*FTM clock prescale value.*
- **ftm\_bdm\_mode\_t bdmMode**  
*FTM behavior in BDM mode.*
- uint32\_t **pwmSyncMode**  
*Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration [ftm\\_pwm\\_sync\\_method\\_t](#).*
- uint32\_t **reloadPoints**  
*FTM reload points; When using this, the PWM synchronization is not required.*
- **ftm\_fault\_mode\_t faultMode**  
*FTM fault control mode.*
- uint8\_t **faultFilterValue**  
*Fault input filter value.*
- **ftm\_deadtime\_prescale\_t deadTimePrescale**  
*The dead time prescalar value.*
- uint32\_t **deadTimeValue**  
*The dead time value deadTimeValue's available range is 0-1023 when register has DTVALEX, otherwise its available range is 0-63.*
- uint32\_t **extTriggers**  
*External triggers to enable.*
- uint8\_t **chnlInitState**

## Enumeration Type Documentation

- `uint8_t chnlPolarity`  
*Defines the initialization value of the channels in OUTINT register.*
- `bool useGlobalTimeBase`  
*Defines the output polarity of the channels in POL register.*  
*True: Use of an external global time base is enabled; False: disabled.*

### 18.5.6.0.0.51 Field Documentation

#### 18.5.6.0.0.51.1 `uint32_t ftm_config_t::pwmSyncMode`

#### 18.5.6.0.0.51.2 `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](#).

#### 18.5.6.0.0.51.3 `uint32_t ftm_config_t::deadTimeValue`

#### 18.5.6.0.0.51.4 `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 Macro Definition Documentation

### 18.6.1 `#define FSL_FTM_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`

## 18.7 Enumeration Type Documentation

### 18.7.1 `enum ftm_chnl_t`

Note

Actual number of available channels is SoC dependent

Enumerator

- `kFTM_Chnl_0` FTM channel number 0.
- `kFTM_Chnl_1` FTM channel number 1.
- `kFTM_Chnl_2` FTM channel number 2.
- `kFTM_Chnl_3` FTM channel number 3.
- `kFTM_Chnl_4` FTM channel number 4.
- `kFTM_Chnl_5` FTM channel number 5.
- `kFTM_Chnl_6` FTM channel number 6.
- `kFTM_Chnl_7` FTM channel number 7.

### 18.7.2 enum ftm\_fault\_input\_t

Enumerator

- kFTM\_Fault\_0* FTM fault 0 input pin.
- kFTM\_Fault\_1* FTM fault 1 input pin.
- kFTM\_Fault\_2* FTM fault 2 input pin.
- kFTM\_Fault\_3* FTM fault 3 input pin.

### 18.7.3 enum ftm\_pwm\_mode\_t

Enumerator

- kFTM\_EdgeAlignedPwm* Edge-aligned PWM.
- kFTM\_CenterAlignedPwm* Center-aligned PWM.
- kFTM\_CombinedPwm* Combined PWM.

### 18.7.4 enum ftm\_pwm\_level\_select\_t

Enumerator

- kFTM\_NoPwmSignal* No PWM output on pin.
- kFTM\_LowTrue* Low true pulses.
- kFTM\_HighTrue* High true pulses.

### 18.7.5 enum ftm\_output\_compare\_mode\_t

Enumerator

- kFTM\_NoOutputSignal* No channel output when counter reaches CnV.
- kFTM\_ToggleOnMatch* Toggle output.
- kFTM\_ClearOnMatch* Clear output.
- kFTM\_SetOnMatch* Set output.

### 18.7.6 enum ftm\_input\_capture\_edge\_t

Enumerator

- kFTM\_RisingEdge* Capture on rising edge only.
- kFTM\_FallingEdge* Capture on falling edge only.
- kFTM\_RiseAndFallEdge* Capture on rising or falling edge.

## Enumeration Type Documentation

### 18.7.7 enum ftm\_dual\_edge\_capture\_mode\_t

Enumerator

*kFTM\_OneShot* One-shot capture mode.

*kFTM\_Continuous* Continuous capture mode.

### 18.7.8 enum ftm\_quad\_decode\_mode\_t

Enumerator

*kFTM\_QuadPhaseEncode* Phase A and Phase B encoding mode.

*kFTM\_QuadCountAndDir* Count and direction encoding mode.

### 18.7.9 enum ftm\_phase\_polarity\_t

Enumerator

*kFTM\_QuadPhaseNormal* Phase input signal is not inverted.

*kFTM\_QuadPhaseInvert* Phase input signal is inverted.

### 18.7.10 enum ftm\_deadtime\_prescale\_t

Enumerator

*kFTM\_Deadtime\_Prescale\_1* Divide by 1.

*kFTM\_Deadtime\_Prescale\_4* Divide by 4.

*kFTM\_Deadtime\_Prescale\_16* Divide by 16.

### 18.7.11 enum ftm\_clock\_source\_t

Enumerator

*kFTM\_SystemClock* System clock selected.

*kFTM\_FixedClock* Fixed frequency clock.

*kFTM\_ExternalClock* External clock.

### 18.7.12 enum ftm\_clock\_prescale\_t

Enumerator

- kFTM\_Prescale\_Divide\_1* Divide by 1.
- kFTM\_Prescale\_Divide\_2* Divide by 2.
- kFTM\_Prescale\_Divide\_4* Divide by 4.
- kFTM\_Prescale\_Divide\_8* Divide by 8.
- kFTM\_Prescale\_Divide\_16* Divide by 16.
- kFTM\_Prescale\_Divide\_32* Divide by 32.
- kFTM\_Prescale\_Divide\_64* Divide by 64.
- kFTM\_Prescale\_Divide\_128* Divide by 128.

### 18.7.13 enum ftm\_bdm\_mode\_t

Enumerator

- kFTM\_BdmMode\_0* FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
- kFTM\_BdmMode\_1* FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
- kFTM\_BdmMode\_2* FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
- kFTM\_BdmMode\_3* FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode.

### 18.7.14 enum ftm\_fault\_mode\_t

Enumerator

- kFTM\_Fault\_Disable* Fault control is disabled for all channels.
- kFTM\_Fault\_EvenChnls* Enabled for even channels only(0,2,4,6) with manual fault clearing.
- kFTM\_Fault\_AllChnlsMan* Enabled for all channels with manual fault clearing.
- kFTM\_Fault\_AllChnlsAuto* Enabled for all channels with automatic fault clearing.

### 18.7.15 enum ftm\_external\_trigger\_t

## Enumeration Type Documentation

Note

Actual available external trigger sources are SoC-specific

Enumerator

- kFTM\_Chnl0Trigger*** Generate trigger when counter equals chnl 0 CnV reg.
- kFTM\_Chnl1Trigger*** Generate trigger when counter equals chnl 1 CnV reg.
- kFTM\_Chnl2Trigger*** Generate trigger when counter equals chnl 2 CnV reg.
- kFTM\_Chnl3Trigger*** Generate trigger when counter equals chnl 3 CnV reg.
- kFTM\_Chnl4Trigger*** Generate trigger when counter equals chnl 4 CnV reg.
- kFTM\_Chnl5Trigger*** Generate trigger when counter equals chnl 5 CnV reg.
- kFTM\_InitTrigger*** Generate Trigger when counter is updated with CNTIN.

### 18.7.16 enum ftm\_pwm\_sync\_method\_t

Enumerator

- kFTM\_SoftwareTrigger*** Software triggers PWM sync.
- kFTM\_HardwareTrigger\_0*** Hardware trigger 0 causes PWM sync.
- kFTM\_HardwareTrigger\_1*** Hardware trigger 1 causes PWM sync.
- kFTM\_HardwareTrigger\_2*** Hardware trigger 2 causes PWM sync.

### 18.7.17 enum ftm\_reload\_point\_t

Note

Actual available reload points are SoC-specific

Enumerator

- kFTM\_Chnl0Match*** Channel 0 match included as a reload point.
- kFTM\_Chnl1Match*** Channel 1 match included as a reload point.
- kFTM\_Chnl2Match*** Channel 2 match included as a reload point.
- kFTM\_Chnl3Match*** Channel 3 match included as a reload point.
- kFTM\_Chnl4Match*** Channel 4 match included as a reload point.
- kFTM\_Chnl5Match*** Channel 5 match included as a reload point.
- kFTM\_Chnl6Match*** Channel 6 match included as a reload point.
- kFTM\_Chnl7Match*** Channel 7 match included as a reload point.
- kFTM\_CntMax*** Use in up-down count mode only, reload when counter reaches the maximum value.

***kFTM\_CntMin*** Use in up-down count mode only, reload when counter reaches the minimum value.

***kFTM\_HalfCycMatch*** Available on certain SoC's, half cycle match reload point.

**18.7.18 enum ftm\_interrupt\_enable\_t**

Note

Actual available interrupts are SoC-specific

Enumerator

- kFTM\_Chnl0InterruptEnable*** Channel 0 interrupt.
- kFTM\_Chnl1InterruptEnable*** Channel 1 interrupt.
- kFTM\_Chnl2InterruptEnable*** Channel 2 interrupt.
- kFTM\_Chnl3InterruptEnable*** Channel 3 interrupt.
- kFTM\_Chnl4InterruptEnable*** Channel 4 interrupt.
- kFTM\_Chnl5InterruptEnable*** Channel 5 interrupt.
- kFTM\_Chnl6InterruptEnable*** Channel 6 interrupt.
- kFTM\_Chnl7InterruptEnable*** Channel 7 interrupt.
- kFTM\_FaultInterruptEnable*** Fault interrupt.
- kFTM\_TimeOverflowInterruptEnable*** Time overflow interrupt.
- kFTM\_ReloadInterruptEnable*** Reload interrupt; Available only on certain SoC's.

**18.7.19 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.

## Function Documentation

### 18.7.20 enum \_ftm\_quad\_decoder\_flags

Enumerator

**kFTM\_QquadDecoderCountingIncreaseFlag** Counting direction is increasing (FTM counter increment), or the direction is decreasing.

**kFTM\_QquadDecoderCountingOverflowOnTopFlag** Indicates if the TOF bit was set on the top or the bottom of counting.

## 18.8 Function Documentation

### 18.8.1 status\_t FTM\_Init ( FTM\_Type \* *base*, const ftm\_config\_t \* *config* )

Note

This API should be called at the beginning of the application which is using the FTM driver.

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.

### 18.8.2 void FTM\_Deinit ( FTM\_Type \* *base* )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

### 18.8.3 void FTM\_GetDefaultConfig ( ftm\_config\_t \* *config* )

The default values are:

```
* config->prescale = kFTM_Prescale_Divide_1;
* config->bdmMode = kFTM_BdmMode_0;
* config->pwmSyncMode = kFTM_SoftwareTrigger;
* config->reloadPoints = 0;
* config->faultMode = kFTM_Fault_Disable;
* config->faultFilterValue = 0;
* config->deadTimePrescale = kFTM_Deadtime_Prescale_1;
* config->deadTimeValue = 0;
```

```
* config->extTriggers = 0;
* config->chnlInitState = 0;
* config->chnlPolarity = 0;
* config->useGlobalTimeBase = false;
*
```

## Parameters

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

#### 18.8.4 **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 <a href="#">ftm_pwm_mode_t</a> |
| <i>pwmFreq_Hz</i>  | PWM signal frequency in Hz                                                          |
| <i>srcClock_Hz</i> | FTM counter clock in Hz                                                             |

## Returns

kStatus\_Success if the PWM setup was successful kStatus\_Error on failure

#### 18.8.5 **void FTM\_UpdatePwmDutycycle ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, ftm\_pwm\_mode\_t *currentPwmMode*, uint8\_t *dutyCyclePercent* )**

## Parameters

## Function Documentation

|                          |                                                                                                                                   |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>              | FTM peripheral base address                                                                                                       |
| <i>chnlNumber</i>        | The channel/channel pair number. In combined mode, this represents the channel pair number                                        |
| <i>currentPwm-Mode</i>   | The current PWM mode set during PWM setup                                                                                         |
| <i>dutyCycle-Percent</i> | New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle) |

### 18.8.6 void FTM\_UpdateChnlEdgeLevelSelect ( FTM\_Type \* *base*, **ftm\_chnl\_t chnlNumber, uint8\_t level** )

Parameters

|                   |                                                                                                                                                   |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | FTM peripheral base address                                                                                                                       |
| <i>chnlNumber</i> | The channel number                                                                                                                                |
| <i>level</i>      | The level to be set to the ELSnB:ELSnA field; Valid values are 00, 01, 10, 11. See the Kinetis SoC reference manual for details about this field. |

### 18.8.7 status\_t FTM\_SetupPwmMode ( FTM\_Type \* *base*, const **ftm\_chnl\_pwm\_config\_param\_t \* chnlParams, uint8\_t numOfChnls, ftm\_pwm\_mode\_t mode** )

Call this function to configure the PWM signal mode, duty cycle in ticks, and edge. Use this function to configure all FTM channels that are used to output a PWM signal. Please note that: This API is similar with [FTM\\_SetupPwm\(\)](#) API, but will not set the timer period, and this API will set channel match value in timer ticks, not period percent.

Parameters

|                   |                                                                                     |
|-------------------|-------------------------------------------------------------------------------------|
| <i>base</i>       | FTM peripheral base address                                                         |
| <i>chnlParams</i> | Array of PWM channel parameters to configure the channel(s)                         |
| <i>numOfChnls</i> | Number of channels to configure; This should be the size of the array passed in     |
| <i>mode</i>       | PWM operation mode, options available in enumeration <a href="#">ftm_pwm_mode_t</a> |

Returns

kStatus\_Success if the PWM setup was successful kStatus\_Error on failure

### 18.8.8 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. |

### 18.8.9 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.                            |

### 18.8.10 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.

## 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>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. |

**18.8.11 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 |

**18.8.12 void FTM\_EnableInterrupts ( FTM\_Type \* *base*, uint32\_t *mask* )**

Parameters

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | FTM peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">ftm_interrupt_enable_t</a> |

**18.8.13 void FTM\_DisableInterrupts ( FTM\_Type \* *base*, uint32\_t *mask* )**

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">ftm_interrupt_enable_t</a> |
|-------------|---------------------------------------------------------------------------------------------------------------------|

**18.8.14 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](#)

**18.8.15 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](#)

**18.8.16 void FTM\_ClearStatusFlags ( FTM\_Type \* *base*, uint32\_t *mask* )**

Parameters

|             |                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | FTM peripheral base address                                                                                      |
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">ftm_status_flags_t</a> |

**18.8.17 static void FTM\_SetTimerPeriod ( FTM\_Type \* *base*, uint32\_t *ticks* )  
[inline], [static]**

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

## Function Documentation

Note

1. This API allows the user to use the FTM module as a timer. Do not mix usage of this API with FTM's PWM setup API's.
2. Call the utility macros provided in the fsl\_common.h to convert usec or msec to ticks.

Parameters

|              |                                                                            |
|--------------|----------------------------------------------------------------------------|
| <i>base</i>  | FTM peripheral base address                                                |
| <i>ticks</i> | A timer period in units of ticks, which should be equal or greater than 1. |

### **18.8.18 static uint32\_t FTM\_GetCurrentTimerCount ( FTM\_Type \* *base* ) [inline], [static]**

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

Call the utility macros provided in the fsl\_common.h to convert ticks to usec or msec.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

Returns

The current counter value in ticks

### **18.8.19 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. |

### **18.8.20 static void FTM\_StopTimer ( FTM\_Type \* *base* ) [inline], [static]**

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

**18.8.21 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 |

**18.8.22 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 |

**18.8.23 static void FTM\_SetGlobalTimeBaseOutputEnable ( FTM\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                  |
|---------------|----------------------------------|
| <i>base</i>   | FTM peripheral base address      |
| <i>enable</i> | true to enable, false to disable |

**18.8.24 static void FTM\_SetOutputMask ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *mask* ) [inline], [static]**

## Function Documentation

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 |

**18.8.25 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 |

**18.8.26 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 |

**18.8.27 static void FTM\_SetComplementaryEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

|                        |                                                                    |
|------------------------|--------------------------------------------------------------------|
| <i>base</i>            | FTM peripheral base address                                        |
| <i>chnlPair-Number</i> | The FTM channel pair number; options are 0, 1, 2, 3                |
| <i>value</i>           | true: enable complementary mode; false: disable complementary mode |

18.8.28 **static void FTM\_SetInvertEnable ( 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 inverting; false: disable inverting    |

**18.8.29 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 |

**18.8.30 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](#).

**18.8.31 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. |

**18.8.32 static uint32\_t FTM\_GetQuadDecoderCounterValue ( FTM\_Type \* *base* )  
[inline], [static]**

Parameters

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

Returns

Current quad Decoder counter value.

**18.8.33 static void FTM\_ClearQuadDecoderCounterValue ( FTM\_Type \* *base* )  
[inline], [static]**

The counter is set as the initial value.

Parameters

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

**18.8.34 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 |

**18.8.35 static void FTM\_SetWriteProtection ( FTM\_Type \* *base*, bool *enable* )  
[inline], [static]**

## Function Documentation

### 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, 3, 1))  
*GPIO driver version 2.3.1.*

### 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, 3, 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 MCUXpresso SDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of MCUXpresso SDK devices.

### 19.5.2 Typical use case

#### 19.5.2.1 Output Operation

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

#### 19.5.2.2 Input Operation

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

## GPIO Configuration

- void [GPIO\\_PinInit](#) (GPIO\_Type \*base, uint32\_t pin, const [gpio\\_pin\\_config\\_t](#) \*config)  
*Initializes a GPIO pin used by the board.*

## GPIO Output Operations

- static void [GPIO\\_PinWrite](#) (GPIO\_Type \*base, uint32\_t pin, uint8\_t output)  
*Sets the output level of the multiple GPIO pins to the logic 1 or 0.*
- static void [GPIO\\_PortSet](#) (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 1.*
- static void [GPIO\\_PortClear](#) (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 0.*
- static void [GPIO\\_PortToggle](#) (GPIO\_Type \*base, uint32\_t mask)  
*Reverses the current output logic of the multiple GPIO pins.*

## GPIO Input Operations

- static uint32\_t [GPIO\\_PinRead](#) (GPIO\_Type \*base, uint32\_t pin)  
*Reads the current input value of the GPIO port.*

## GPIO Interrupt

- uint32\_t [GPIO\\_PortGetInterruptFlags](#) (GPIO\_Type \*base)  
*Reads the GPIO port interrupt status flag.*

## GPIO Driver

- void [GPIO\\_PortClearInterruptFlags](#) (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\\_PinWrite](#) ( GPIO\_Type \* *base*, uint32\_t *pin*, uint8\_t *output* ) [inline], [static]

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>pin</i>  | GPIO pin number                                                |

|               |                                                                                                                                                                                     |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>output</i> | GPIO pin output logic level. <ul style="list-style-type: none"><li>• 0: corresponding pin output low-logic level.</li><li>• 1: corresponding pin output high-logic level.</li></ul> |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 19.5.3.3 static void GPIO\_PortSet ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>mask</i> | GPIO pin number macro                                          |

### 19.5.3.4 static void GPIO\_PortClear ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>mask</i> | GPIO pin number macro                                          |

### 19.5.3.5 static void GPIO\_PortToggle ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>mask</i> | GPIO pin number macro                                          |

### 19.5.3.6 static **uint32\_t** GPIO\_PinRead ( **GPIO\_Type** \* *base*, **uint32\_t** *pin* ) [inline], [static]

Parameters

## GPIO Driver

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>pin</i>  | GPIO pin number                                                |

Return values

|             |                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>GPIO</i> | port input value <ul style="list-style-type: none"><li>• 0: corresponding pin input low-logic level.</li><li>• 1: corresponding pin input high-logic level.</li></ul> |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 19.5.3.7 `uint32_t GPIO_PortGetInterruptFlags ( GPIO_Type * base )`

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
|-------------|----------------------------------------------------------------|

Return values

|            |                                                                                                             |
|------------|-------------------------------------------------------------------------------------------------------------|
| <i>The</i> | current GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt. |
|------------|-------------------------------------------------------------------------------------------------------------|

### 19.5.3.8 `void GPIO_PortClearInterruptFlags ( GPIO_Type * base, uint32_t mask )`

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>mask</i> | GPIO pin number macro                                          |

## 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

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

#### 19.6.1.2 Input Operation

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



# Chapter 20

## I2C: Inter-Integrated Circuit Driver

### 20.1 Overview

#### Modules

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver

## I2C Driver

### 20.2 I2C Driver

#### 20.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of MCUXpresso SDK devices.

The I2C driver includes functional APIs and transactional APIs.

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

Transactional APIs target the high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code using the functional APIs or accessing the hardware registers.

Transactional APIs support asynchronous transfer. This means that the functions [I2C\\_MasterTransferNonBlocking\(\)](#) set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

#### 20.2.2 Typical use case

##### 20.2.2.1 Master Operation in functional method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/i2c

##### 20.2.2.2 Master Operation in interrupt transactional method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/i2c

##### 20.2.2.3 Master Operation in DMA transactional method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/i2c

##### 20.2.2.4 Slave Operation in functional method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/i2c

##### 20.2.2.5 Slave Operation in interrupt transactional method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/i2c

## 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...](#)

## Macros

- #define `I2C_WAIT_TIMEOUT` 0U /\* Define to zero means keep waiting until the flag is asserted/deasserted. \*/  
*Timeout times for waiting flag.*

## 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` {

## I2C Driver

```
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\_SlaveGeneralCallEvent = 0x40U,  
  kI2C\_SlaveAllEvents }

*Set of events sent to the callback for nonblocking slave transfers.*

## Variables

- I2C\_Type \*const `s_i2cBases` []  
*Pointers to i2c bases for each instance.*

## Driver version

- #define **FSL\_I2C\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 6))  
*I2C driver version 2.0.6.*

## Initialization and deinitialization

- void **I2C\_MasterInit** (I2C\_Type \*base, const i2c\_master\_config\_t \*masterConfig, uint32\_t srcClock\_Hz)  
*Initializes the I2C peripheral.*
- void **I2C\_SlaveInit** (I2C\_Type \*base, const i2c\_slave\_config\_t \*slaveConfig, uint32\_t srcClock\_Hz)  
*Initializes the I2C peripheral.*
- void **I2C\_MasterDeinit** (I2C\_Type \*base)  
*De-initializes the I2C master peripheral.*
- void **I2C\_SlaveDeinit** (I2C\_Type \*base)  
*De-initializes the I2C slave peripheral.*
- uint32\_t **I2CGetInstance** (I2C\_Type \*base)  
*Get instance number for I2C module.*
- void **I2C\_MasterGetDefaultConfig** (i2c\_master\_config\_t \*masterConfig)  
*Sets the I2C master configuration structure to default values.*
- void **I2C\_SlaveGetDefaultConfig** (i2c\_slave\_config\_t \*slaveConfig)  
*Sets the I2C slave configuration structure to default values.*
- static void **I2C\_Enable** (I2C\_Type \*base, bool enable)  
*Enables or disables the I2C peripheral operation.*

## Status

- uint32\_t **I2C\_MasterGetStatusFlags** (I2C\_Type \*base)  
*Gets the I2C status flags.*
- 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.*

## I2C Driver

### 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.*
- status\_t [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.*
- status\_t [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 [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.*

#### 20.2.3.1.0.52 Field Documentation

##### 20.2.3.1.0.52.1 bool i2c\_master\_config\_t::enableMaster

##### 20.2.3.1.0.52.2 bool i2c\_master\_config\_t::enableStopHold

##### 20.2.3.1.0.52.3 uint32\_t i2c\_master\_config\_t::baudRate\_Bps

##### 20.2.3.1.0.52.4 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 [enableBaudRateCtl](#)  
*Enables/disables independent slave baud rate on SCL in very fast I2C modes.*
- uint16\_t [slaveAddress](#)  
*A slave address configuration.*

## I2C Driver

- `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.*

### 20.2.3.2.0.53 Field Documentation

20.2.3.2.0.53.1 `bool i2c_slave_config_t::enableSlave`

20.2.3.2.0.53.2 `bool i2c_slave_config_t::enableGeneralCall`

20.2.3.2.0.53.3 `bool i2c_slave_config_t::enableWakeUp`

20.2.3.2.0.53.4 `bool i2c_slave_config_t::enableBaudRateCtl`

20.2.3.2.0.53.5 `uint16_t i2c_slave_config_t::slaveAddress`

20.2.3.2.0.53.6 `uint16_t i2c_slave_config_t::upperAddress`

20.2.3.2.0.53.7 `i2c_slave_address_mode_t i2c_slave_config_t::addressingMode`

20.2.3.2.0.53.8 `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.*

### 20.2.3.3.0.54 Field Documentation

20.2.3.3.0.54.1 `uint32_t i2c_master_transfer_t::flags`

20.2.3.3.0.54.2 `uint8_t i2c_master_transfer_t::slaveAddress`

20.2.3.3.0.54.3 `i2c_direction_t i2c_master_transfer_t::direction`

20.2.3.3.0.54.4 `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

20.2.3.3.0.54.5 `uint8_t i2c_master_transfer_t::subaddressSize`

20.2.3.3.0.54.6 `uint8_t* volatile i2c_master_transfer_t::data`

20.2.3.3.0.54.7 `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.*

## I2C Driver

### 20.2.3.4.0.55 Field Documentation

20.2.3.4.0.55.1 `i2c_master_transfer_t i2c_master_handle_t::transfer`

20.2.3.4.0.55.2 `size_t i2c_master_handle_t::transferSize`

20.2.3.4.0.55.3 `uint8_t i2c_master_handle_t::state`

20.2.3.4.0.55.4 `i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback`

20.2.3.4.0.55.5 `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.*

### 20.2.3.5.0.56 Field Documentation

20.2.3.5.0.56.1 `i2c_slave_transfer_event_t i2c_slave_transfer_t::event`

20.2.3.5.0.56.2 `uint8_t* volatile i2c_slave_transfer_t::data`

20.2.3.5.0.56.3 `volatile size_t i2c_slave_transfer_t::dataSize`

20.2.3.5.0.56.4 `status_t i2c_slave_transfer_t::completionStatus`

Only applies for `kI2C_SlaveCompletionEvent`.

20.2.3.5.0.56.5 `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.*

### 20.2.3.6.0.57 Field Documentation

20.2.3.6.0.57.1 **volatile bool i2c\_slave\_handle\_t::isBusy**

20.2.3.6.0.57.2 **i2c\_slave\_transfer\_t i2c\_slave\_handle\_t::transfer**

20.2.3.6.0.57.3 **uint32\_t i2c\_slave\_handle\_t::eventMask**

20.2.3.6.0.57.4 **i2c\_slave\_transfer\_callback\_t i2c\_slave\_handle\_t::callback**

20.2.3.6.0.57.5 **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, 6))**

20.2.4.2 **#define I2C\_WAIT\_TIMEOUT 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/**

### 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** Timeout poling status flags.

**kStatus\_I2C\_Addr\_Nak** NAK received during the address probe.

## I2C Driver

### 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.

**kI2C\_TransferNoStartFlag** A transfer starts without a start signal, only support write only or write+read with no start flag, do not support read only with no start flag.

**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.

## I2C Driver

### 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.

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 uint32\_t I2CGetInstance ( I2C\_Type \* *base* )

Parameters

|             |                              |
|-------------|------------------------------|
| <i>base</i> | I2C peripheral base address. |
|-------------|------------------------------|

#### 20.2.7.6 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);
*
```

## I2C Driver

Parameters

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <i>masterConfig</i> | A pointer to the master configuration structure. |
|---------------------|--------------------------------------------------|

### 20.2.7.7 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. |
|--------------------|-------------------------------------------------|

### 20.2.7.8 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. |

### 20.2.7.9 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.

### 20.2.7.10 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.

#### 20.2.7.11 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.

Parameters

|                   |                                                                                                                                                                                                                                                                                                                                            |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | I2C base pointer                                                                                                                                                                                                                                                                                                                           |
| <i>statusMask</i> | <p>The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> <li>• kI2C_StartDetectFlag (if available)</li> <li>• kI2C_StopDetectFlag (if available)</li> <li>• kI2C_ArbitrationLostFlag</li> <li>• kI2C_IntPendingFlagFlag</li> </ul> |

#### 20.2.7.12 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> | <p>The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values:</p> <ul style="list-style-type: none"> <li>• kI2C_StartDetectFlag (if available)</li> <li>• kI2C_StopDetectFlag (if available)</li> <li>• kI2C_ArbitrationLostFlag</li> <li>• kI2C_IntPendingFlagFlag</li> </ul> |

#### 20.2.7.13 void I2C\_EnableInterrupts ( I2C\_Type \* *base*, uint32\_t *mask* )

## I2C Driver

Parameters

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | I2C base pointer                                                                                                                                                                                                                                                                     |
| <i>mask</i> | interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"><li>• kI2C_GlobalInterruptEnable</li><li>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</li><li>• kI2C_SdaTimeoutInterruptEnable</li></ul> |

### 20.2.7.14 void I2C\_DisableInterrupts ( I2C\_Type \* *base*, uint32\_t *mask* )

Parameters

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | I2C base pointer                                                                                                                                                                                                                                                                     |
| <i>mask</i> | interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"><li>• kI2C_GlobalInterruptEnable</li><li>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</li><li>• kI2C_SdaTimeoutInterruptEnable</li></ul> |

### 20.2.7.15 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 |

### 20.2.7.16 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

#### 20.2.7.17 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               |

#### 20.2.7.18 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.                |

#### 20.2.7.19 status\_t I2C\_MasterStop ( I2C\_Type \* *base* )

## I2C Driver

Return values

|                            |                                    |
|----------------------------|------------------------------------|
| <i>kStatus_Success</i>     | Successfully send the stop signal. |
| <i>kStatus_I2C_Timeout</i> | Send stop signal failed, timeout.  |

### 20.2.7.20 **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. |

### 20.2.7.21 **status\_t I2C\_MasterWriteBlocking ( I2C\_Type \* *base*, const uint8\_t \* *txBuff*, size\_t *txSize*, uint32\_t *flags* )**

Parameters

|               |                                                                                                                                                       |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | The I2C peripheral base pointer.                                                                                                                      |
| <i>txBuff</i> | The pointer to the data to be transferred.                                                                                                            |
| <i>txSize</i> | The length in bytes of the data to be transferred.                                                                                                    |
| <i>flags</i>  | Transfer control flag to decide whether need to send a stop, use kI2C_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.            |

|                              |                                              |
|------------------------------|----------------------------------------------|
| <code>kStatus_I2C_Nak</code> | Transfer error, receive NAK during transfer. |
|------------------------------|----------------------------------------------|

### 20.2.7.22 `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

|                     |                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>base</code>   | I2C peripheral base pointer.                                                                                                                                                    |
| <code>rxBuff</code> | The pointer to the data to store the received data.                                                                                                                             |
| <code>rxSize</code> | The length in bytes of the data to be received.                                                                                                                                 |
| <code>flags</code>  | Transfer control flag to decide whether need to send a stop, use <code>kI2C_TransferDefaultFlag</code> to issue a stop and <code>kI2C_TransferNoStop</code> to not send a stop. |

Return values

|                                  |                                              |
|----------------------------------|----------------------------------------------|
| <code>kStatus_Success</code>     | Successfully complete the data transmission. |
| <code>kStatus_I2C_Timeout</code> | Send stop signal failed, timeout.            |

### 20.2.7.23 `status_t I2C_SlaveWriteBlocking ( I2C_Type * base, const uint8_t * txBuff, size_t txSize )`

Parameters

|                     |                                                    |
|---------------------|----------------------------------------------------|
| <code>base</code>   | The I2C peripheral base pointer.                   |
| <code>txBuff</code> | The pointer to the data to be transferred.         |
| <code>txSize</code> | The length in bytes of the data to be transferred. |

Return values

## I2C Driver

|                                     |                                              |
|-------------------------------------|----------------------------------------------|
| <i>kStatus_Success</i>              | Successfully complete the data transmission. |
| <i>kStatus_I2C_Arbitration-Lost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer. |

### 20.2.7.24 **status\_t 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.     |

Return values

|                            |                                     |
|----------------------------|-------------------------------------|
| <i>kStatus_Success</i>     | Successfully complete data receive. |
| <i>kStatus_I2C_Timeout</i> | Wait status flag timeout.           |

### 20.2.7.25 **status\_t I2C\_MasterTransferBlocking ( I2C\_Type \* *base*, i2c\_master\_transfer\_t \* *xfer* )**

Note

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

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | I2C peripheral base address.       |
| <i>xfer</i> | Pointer to the transfer structure. |

Return values

|                        |                                              |
|------------------------|----------------------------------------------|
| <i>kStatus_Success</i> | Successfully complete the data transmission. |
|------------------------|----------------------------------------------|

|                                     |                                              |
|-------------------------------------|----------------------------------------------|
| <i>kStatus_I2C_Busy</i>             | Previous transmission still not finished.    |
| <i>kStatus_I2C_Timeout</i>          | Transfer error, wait signal timeout.         |
| <i>kStatus_I2C_Arbitration-Lost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer. |

### 20.2.7.26 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.                              |

### 20.2.7.27 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. |
|------------------------|-------------------------------------------|

## I2C Driver

|                            |                                           |
|----------------------------|-------------------------------------------|
| <i>kStatus_I2C_Busy</i>    | Previous transmission still not finished. |
| <i>kStatus_I2C_Timeout</i> | Transfer error, wait signal timeout.      |

### 20.2.7.28 **status\_t I2C\_MasterTransferGetCount ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|               |                                                                           |
|---------------|---------------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                         |
| <i>handle</i> | pointer to 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. |

### 20.2.7.29 **status\_t 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 |

Return values

|                            |                                  |
|----------------------------|----------------------------------|
| <i>kStatus_I2C_Timeout</i> | Timeout during polling flag.     |
| <i>kStatus_Success</i>     | Successfully abort the transfer. |

### 20.2.7.30 **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. |

#### 20.2.7.31 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.                      |

#### 20.2.7.32 status\_t I2C\_SlaveTransferNonBlocking ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, uint32\_t *eventMask* )

Call this API after calling the [I2C\\_SlaveInit\(\)](#) and [I2C\\_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to [I2C\\_SlaveTransferCreateHandle\(\)](#). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The [k\\_I2C\\_SlaveTransmitEvent](#) and #[kLPI2C\\_SlaveReceiveEvent](#) events are always enabled and do not need to be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [kI2C\\_SlaveAllEvents](#) constant is provided as a convenient way to enable all events.

Parameters

|                  |                                                                                                                                                                                                                                                                                                    |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>      | The I2C peripheral base address.                                                                                                                                                                                                                                                                   |
| <i>handle</i>    | Pointer to # <a href="#">i2c_slave_handle_t</a> structure which stores the transfer state.                                                                                                                                                                                                         |
| <i>eventMask</i> | Bit mask formed by OR'ing together <a href="#">i2c_slave_transfer_event_t</a> enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and <a href="#">kI2C_SlaveAllEvents</a> to enable all events. |

## I2C Driver

Return values

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| #kStatus_Success | Slave transfers were successfully started.                |
| kStatus_I2C_Busy | Slave transfers have already been started on this handle. |

### 20.2.7.33 void I2C\_SlaveTransferAbort ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle* )

Note

This API can be called at any time to stop slave for handling the bus events.

Parameters

|               |                                                                          |
|---------------|--------------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                        |
| <i>handle</i> | pointer to i2c_slave_handle_t structure which stores the transfer state. |

### 20.2.7.34 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

|                         |                                |
|-------------------------|--------------------------------|
| kStatus_InvalidArgument | count is Invalid.              |
| kStatus_Success         | Successfully return the count. |

### 20.2.7.35 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 |

## 20.2.8 Variable Documentation

### 20.2.8.1 I2C\_Type\* const s\_i2cBases[]

## I2C eDMA Driver

### 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.*

#### Driver version

- #define [FSL\\_I2C\\_EDMA\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 5))  
*I2C EDMA driver version 2.0.5.*

### 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 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.*

### 20.3.2.1.0.58 Field Documentation

20.3.2.1.0.58.1 **i2c\_master\_transfer\_t i2c\_master\_edma\_handle\_t::transfer**

20.3.2.1.0.58.2 **size\_t i2c\_master\_edma\_handle\_t::transferSize**

20.3.2.1.0.58.3 **uint8\_t i2c\_master\_edma\_handle\_t::nbytes**

20.3.2.1.0.58.4 **uint8\_t i2c\_master\_edma\_handle\_t::state**

20.3.2.1.0.58.5 **edma\_handle\_t\* i2c\_master\_edma\_handle\_t::dmaHandle**

20.3.2.1.0.58.6 **i2c\_master\_edma\_transfer\_callback\_t i2c\_master\_edma\_handle\_t::completionCallback**

20.3.2.1.0.58.7 **void\* i2c\_master\_edma\_handle\_t::userData**

### 20.3.3 Macro Definition Documentation

20.3.3.1 **#define FSL\_I2C\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 5))**

### 20.3.4 Typedef Documentation

20.3.4.1 **typedef void(\* i2c\_master\_edma\_transfer\_callback\_t)(I2C\_Type \*base, i2c\_master\_edma\_handle\_t \*handle, status\_t status, void \*userData)**

### 20.3.5 Function Documentation

20.3.5.1 **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 )**

## I2C eDMA Driver

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.                                 |

**20.3.5.2 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 i2c_master_edma_handle_t structure.                           |
| <i>xfer</i>   | A pointer to the transfer structure of <a href="#">i2c_master_transfer_t</a> . |

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.   |

**20.3.5.3 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 i2c_master_edma_handle_t structure.           |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction. |

#### 20.3.5.4 void I2C\_MasterTransferAbortEDMA ( I2C\_Type \* *base*, i2c\_master\_edma\_handle\_t \* *handle* )

Parameters

|               |                                                      |
|---------------|------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address.                         |
| <i>handle</i> | A pointer to the i2c_master_edma_handle_t structure. |

## I2C DMA Driver

### 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.*

#### Driver version

- #define [FSL\\_I2C\\_DMA\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 5))  
*I2C DMA driver version 2.0.5.*

## 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**  
*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.*

#### 20.4.2.1.0.59 Field Documentation

20.4.2.1.0.59.1 **i2c\_master\_transfer\_t i2c\_master\_dma\_handle\_t::transfer**

20.4.2.1.0.59.2 **size\_t i2c\_master\_dma\_handle\_t::transferSize**

20.4.2.1.0.59.3 **uint8\_t i2c\_master\_dma\_handle\_t::state**

20.4.2.1.0.59.4 **dma\_handle\_t\* i2c\_master\_dma\_handle\_t::dmaHandle**

20.4.2.1.0.59.5 **i2c\_master\_dma\_transfer\_callback\_t i2c\_master\_dma\_handle\_t::completionCallback**

20.4.2.1.0.59.6 **void\* i2c\_master\_dma\_handle\_t::userData**

#### 20.4.3 Macro Definition Documentation

20.4.3.1 **#define FSL\_I2C\_DMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 5))**

#### 20.4.4 Typedef Documentation

20.4.4.1 **typedef void(\* i2c\_master\_dma\_transfer\_callback\_t)(I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, status\_t status, void \*userData)**

#### 20.4.5 Function Documentation

20.4.5.1 **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 )**

## I2C DMA Driver

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                               |

### 20.4.5.2 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>kStataus_I2C_Nak</i>             | A transfer error, receives NAK during transfer. |

### 20.4.5.3 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. |
|--------------|-----------------------------------------------------------------------|

#### 20.4.5.4 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

##### Driver version

- #define `FSL_I2C_FREERTOS_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 5))  
*I2C freertos driver version 2.0.5.*

##### 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 Macro Definition Documentation

##### 20.5.2.1 #define `FSL_I2C_FREERTOS_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 5))

#### 20.5.3 Function Documentation

##### 20.5.3.1 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.

### 20.5.3.2 status\_t I2C\_RTOS\_Deinit( i2c\_rtos\_handle\_t \* handle )

This function deinitializes the I2C module and the related RTOS context.

## I2C FreeRTOS Driver

Parameters

|               |                      |
|---------------|----------------------|
| <i>handle</i> | The RTOS I2C handle. |
|---------------|----------------------|

### 20.5.3.3 **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 MCUXpresso SDK provides a peripheral driver for the Low-Leakage Wakeup Unit (LLWU) module of MCUXpresso SDK devices. The LLWU module allows the user to select external pin sources and internal modules as a wake-up source from low-leakage power modes.

### 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.*

## Macro Definition Documentation

### Driver version

- #define **FSL\_LLWU\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 2))  
*LLWU driver version 2.0.2.*

## 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.*
- void **LLWU\_SetResetPinMode** (LLWU\_Type \*base, bool pinEnable, bool pinFilterEnable)  
*Sets the reset pin mode.*
- #define **INTERNAL\_WAKEUP\_MODULE\_FLAG\_REG** F3

## 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, 2))

## 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.

## Function Documentation

Parameters

|                 |                                   |
|-----------------|-----------------------------------|
| <i>base</i>     | LLWU peripheral base address.     |
| <i>pinIndex</i> | A pin index, which starts from 1. |

Returns

True if the specific pin is a wakeup source.

### 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.

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.

## Function Documentation

Parameters

|                    |                                                        |
|--------------------|--------------------------------------------------------|
| <i>base</i>        | LLWU peripheral base address.                          |
| <i>filterIndex</i> | A pin filter index to clear the flag, starting from 1. |

### 21.8.9 void LLWU\_SetResetPinMode ( LLWU\_Type \* *base*, bool *pinEnable*, bool *pinFilterEnable* )

This function determines how the reset pin is used as a low leakage mode exit source.

Parameters

|                         |                                                                      |
|-------------------------|----------------------------------------------------------------------|
| <i>pinEnable</i>        | Enable reset the pin filter                                          |
| <i>pinFilter-Enable</i> | Specify whether the pin filter is enabled in Low-Leakage power mode. |

# Chapter 22

## LPTMR: Low-Power Timer

### 22.1 Overview

The MCUXpresso SDK provides a driver for the Low-Power Timer (LPTMR) of MCUXpresso SDK devices.

### 22.2 Function groups

The LPTMR driver supports operating the module as a time counter or as a pulse counter.

#### 22.2.1 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.

#### 22.2.2 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.

#### 22.2.3 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

### 22.2.4 Status

Provides functions to get and clear the LPTMR status.

### 22.2.5 Interrupt

Provides functions to enable/disable LPTMR interrupts and get the currently enabled interrupts.

## 22.3 Typical use case

### 22.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lptmr

## Data Structures

- struct [lptmr\\_config\\_t](#)  
*LPTMR config structure.* [More...](#)

## Enumerations

- enum [lptmr\\_pin\\_select\\_t](#) {  
  kLPTMR\_PinSelectInput\_0 = 0x0U,  
  kLPTMR\_PinSelectInput\_1 = 0x1U,  
  kLPTMR\_PinSelectInput\_2 = 0x2U,  
  kLPTMR\_PinSelectInput\_3 = 0x3U }  
*LPTMR pin selection used in pulse counter mode.*
- enum [lptmr\\_pin\\_polarity\\_t](#) {  
  kLPTMR\_PinPolarityActiveHigh = 0x0U,  
  kLPTMR\_PinPolarityActiveLow = 0x1U }  
*LPTMR pin polarity used in pulse counter mode.*
- enum [lptmr\\_timer\\_mode\\_t](#) {  
  kLPTMR\_TimerModeTimeCounter = 0x0U,  
  kLPTMR\_TimerModePulseCounter = 0x1U }  
*LPTMR timer mode selection.*
- enum [lptmr\\_prescaler\\_glitch\\_value\\_t](#) {

```

kLPTMR_Prescale_Glitch_0 = 0x0U,
kLPTMR_Prescale_Glitch_1 = 0x1U,
kLPTMR_Prescale_Glitch_2 = 0x2U,
kLPTMR_Prescale_Glitch_3 = 0x3U,
kLPTMR_Prescale_Glitch_4 = 0x4U,
kLPTMR_Prescale_Glitch_5 = 0x5U,
kLPTMR_Prescale_Glitch_6 = 0x6U,
kLPTMR_Prescale_Glitch_7 = 0x7U,
kLPTMR_Prescale_Glitch_8 = 0x8U,
kLPTMR_Prescale_Glitch_9 = 0x9U,
kLPTMR_Prescale_Glitch_10 = 0xAU,
kLPTMR_Prescale_Glitch_11 = 0xBU,
kLPTMR_Prescale_Glitch_12 = 0xCU,
kLPTMR_Prescale_Glitch_13 = 0xDU,
kLPTMR_Prescale_Glitch_14 = 0xEU,
kLPTMR_Prescale_Glitch_15 = 0xFU }

```

*LPTMR prescaler/glitch filter values.*

- enum `lptmr_prescaler_clock_select_t` {
   
kLPTMR\_PrescalerClock\_0 = 0x0U,
 kLPTMR\_PrescalerClock\_1 = 0x1U,
 kLPTMR\_PrescalerClock\_2 = 0x2U,
 kLPTMR\_PrescalerClock\_3 = 0x3U }

*LPTMR prescaler/glitch filter clock select.*

- enum `lptmr_interrupt_enable_t` { `kLPTMR_TimerInterruptEnable` = LPTMR\_CSR\_TIE\_MASK }
- List of the LPTMR interrupts.*
- enum `lptmr_status_flags_t` { `kLPTMR_TimerCompareFlag` = LPTMR\_CSR\_TCF\_MASK }
- List of the LPTMR status flags.*

## Driver version

- #define `FSL_LPTMR_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 1))  
*Version 2.0.1.*

## Initialization and deinitialization

- void `LPTMR_Init` (LPTMR\_Type \*base, const `lptmr_config_t` \*config)  
*Ungates the LPTMR clock and configures the peripheral for a basic operation.*
- void `LPTMR_Deinit` (LPTMR\_Type \*base)  
*Gates the LPTMR clock.*
- void `LPTMR_GetDefaultConfig` (`lptmr_config_t` \*config)  
*Fills in the LPTMR configuration structure with default settings.*

## Interrupt Interface

- static void `LPTMR_EnableInterrupts` (LPTMR\_Type \*base, uint32\_t mask)  
*Enables the selected LPTMR interrupts.*
- static void `LPTMR_DisableInterrupts` (LPTMR\_Type \*base, uint32\_t mask)  
*Disables the selected LPTMR interrupts.*

## Data Structure Documentation

- static uint32\_t [LPTMR\\_GetEnabledInterrupts](#) (LPTMR\_Type \*base)  
*Gets the enabled LPTMR interrupts.*

## Status Interface

- static uint32\_t [LPTMR\\_GetStatusFlags](#) (LPTMR\_Type \*base)  
*Gets the LPTMR status flags.*
- static void [LPTMR\\_ClearStatusFlags](#) (LPTMR\_Type \*base, uint32\_t mask)  
*Clears the LPTMR status flags.*

## Read and write the timer period

- static void [LPTMR\\_SetTimerPeriod](#) (LPTMR\_Type \*base, uint32\_t ticks)  
*Sets the timer period in units of count.*
- static uint32\_t [LPTMR\\_GetCurrentTimerCount](#) (LPTMR\_Type \*base)  
*Reads the current timer counting value.*

## Timer Start and Stop

- static void [LPTMR\\_StartTimer](#) (LPTMR\_Type \*base)  
*Starts the timer.*
- static void [LPTMR\\_StopTimer](#) (LPTMR\_Type \*base)  
*Stops the timer.*

## 22.4 Data Structure Documentation

### 22.4.1 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\_prescaler\_glitch\_value\_t** value  
*Prescaler or glitch filter value.*

## 22.5 Enumeration Type Documentation

### 22.5.1 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.

### 22.5.2 enum lptmr\_pin\_polarity\_t

Enumerator

- kLPTMR\_PinPolarityActiveHigh** Pulse Counter input source is active-high.
- kLPTMR\_PinPolarityActiveLow** Pulse Counter input source is active-low.

### 22.5.3 enum lptmr\_timer\_mode\_t

Enumerator

- kLPTMR\_TimerModeTimeCounter** Time Counter mode.
- kLPTMR\_TimerModePulseCounter** Pulse Counter mode.

### 22.5.4 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.

## Function Documentation

- 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.

### 22.5.5 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.

### 22.5.6 enum lptmr\_interrupt\_enable\_t

Enumerator

*kLPTMR\_TimerInterruptEnable* Timer interrupt enable.

### 22.5.7 enum lptmr\_status\_flags\_t

Enumerator

*kLPTMR\_TimerCompareFlag* Timer compare flag.

## 22.6 Function Documentation

### 22.6.1 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. |

## 22.6.2 void LPTMR\_Deinit ( LPTMR\_Type \* *base* )

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

## 22.6.3 void LPTMR\_GetDefaultConfig ( lptmr\_config\_t \* *config* )

The default values are as follows.

```
* config->timerMode = kLPTMR_TimerModeTimeCounter;
* config->pinSelect = kLPTMR_PinSelectInput_0;
* config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
* config->enableFreeRunning = false;
* config->bypassPrescaler = true;
* config->prescalerClockSource = kLPTMR_PrescalerClock_1;
* config->value = kLPTMR_Prescale_Glitch_0;
*
```

Parameters

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

## 22.6.4 static void LPTMR\_EnableInterrupts ( LPTMR\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">lptmr_interrupt_enable_t</a> |

## 22.6.5 static void LPTMR\_DisableInterrupts ( LPTMR\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

## Function Documentation

Parameters

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                            |
| <i>mask</i> | The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">lptmr_interrupt_enable_t</a> . |

**22.6.6 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](#)

**22.6.7 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](#)

**22.6.8 static void LPTMR\_ClearStatusFlags ( LPTMR\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                        |
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">lptmr_status_flags_t</a> . |

### 22.6.9 static void LPTMR\_SetTimerPeriod ( LPTMR\_Type \* *base*, uint32\_t *ticks* ) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note

1. The TCF flag is set with the CNR equals the count provided here and then increments.
2. Call the utility macros provided in the `fsl_common.h` to convert to ticks.

Parameters

|              |                                                                            |
|--------------|----------------------------------------------------------------------------|
| <i>base</i>  | LPTMR peripheral base address                                              |
| <i>ticks</i> | A timer period in units of ticks, which should be equal or greater than 1. |

### 22.6.10 static uint32\_t LPTMR\_GetCurrentTimerCount ( LPTMR\_Type \* *base* ) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

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

## Function Documentation

### 22.6.11 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 |
|-------------|-------------------------------|

### 22.6.12 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 23

## PDB: Programmable Delay Block

### 23.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Programmable Delay Block (PDB) module of MCUXpresso SDK devices.

The PDB driver includes a basic PDB counter, trigger generators for ADC, DAC, and pulse-out.

The basic PDB counter can be used as a general programmable timer with an interrupt. The counter increases automatically with the divided clock signal after it is triggered to start by an external trigger input or the software trigger. There are "milestones" for the output trigger event. When the counter is equal to any of these "milestones", the corresponding trigger is generated and sent out to other modules. These "milestones" are for the following events.

- Counter delay interrupt, which is the interrupt for the PDB module
- ADC pre-trigger to trigger the ADC conversion
- DAC interval trigger to trigger the DAC buffer and move the buffer read pointer
- Pulse-out triggers to generate a single 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.

### 23.2 Typical use case

#### 23.2.1 Working as basic PDB counter with a PDB interrupt.

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

#### 23.2.2 Working with an additional trigger. The ADC trigger is used as an example.

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

### 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...](#)*

## Typical use case

### 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)`,  
  `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.*

- enum `pdb_adc_trigger_channel_t` {
 

```
kPDB_ADCTriggerChannel0 = 0U,
kPDB_ADCTriggerChannel1 = 1U,
kPDB_ADCTriggerChannel2 = 2U,
kPDB_ADCTriggerChannel3 = 3U }
```

*List of PDB ADC trigger channels.*

- enum `pdb_adc_pretrigger_t` {
 

```
kPDB_ADCPreTrigger0 = 0U,
kPDB_ADCPreTrigger1 = 1U,
kPDB_ADCPreTrigger2 = 2U,
kPDB_ADCPreTrigger3 = 3U,
kPDB_ADCPreTrigger4 = 4U,
kPDB_ADCPreTrigger5 = 5U,
kPDB_ADCPreTrigger6 = 6U,
kPDB_ADCPreTrigger7 = 7U }
```

*List of PDB ADC pretrigger.*

- enum `pdb_dac_trigger_channel_t` {
 

```
kPDB_DACTriggerChannel0 = 0U,
kPDB_DACTriggerChannel1 = 1U }
```

*List of PDB DAC trigger channels.*

- enum `pdb_pulse_out_trigger_channel_t` {
 

```
kPDB_PulseOutTriggerChannel0 = 0U,
kPDB_PulseOutTriggerChannel1 = 1U,
kPDB_PulseOutTriggerChannel2 = 2U,
kPDB_PulseOutTriggerChannel3 = 3U }
```

*List of PDB pulse out trigger channels.*

- enum `pdb_pulse_out_channel_mask_t` {

## Typical use case

```
kPDB_PulseOutChannel0Mask = (1U << 0U),
kPDB_PulseOutChannel1Mask = (1U << 1U),
kPDB_PulseOutChannel2Mask = (1U << 2U),
kPDB_PulseOutChannel3Mask = (1U << 3U) }
```

*List of PDB pulse out trigger channels mask.*

## 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, **pdb\_adc\_trigger\_channel\_t** channel, **pdb\_adc\_pretrigger\_config\_t** \*config)  
*Configures the ADC pre-trigger in the PDB module.*

- static void [PDB\\_SetADCPreTriggerDelayValue](#) (PDB\_Type \*base, [pdb\\_adc\\_trigger\\_channel\\_t](#) channel, [pdb\\_adc\\_pretrigger\\_t](#) pretriggerNumber, uint32\_t value)  
*Sets the value for the ADC pre-trigger delay event.*
- static uint32\_t [PDB\\_GetADCPreTriggerStatusFlags](#) (PDB\_Type \*base, [pdb\\_adc\\_trigger\\_channel\\_t](#) channel)  
*Gets the ADC pre-trigger's status flags.*
- static void [PDB\\_ClearADCPreTriggerStatusFlags](#) (PDB\_Type \*base, [pdb\\_adc\\_trigger\\_channel\\_t](#) channel, uint32\_t mask)  
*Clears the ADC pre-trigger status flags.*

## DAC Interval Trigger

- void [PDB\\_SetDACTriggerConfig](#) (PDB\_Type \*base, [pdb\\_dac\\_trigger\\_channel\\_t](#) channel, [pdb\\_dac\\_trigger\\_config\\_t](#) \*config)  
*Configures the DAC trigger in the PDB module.*
- static void [PDB\\_SetDACTriggerIntervalValue](#) (PDB\_Type \*base, [pdb\\_dac\\_trigger\\_channel\\_t](#) channel, uint32\_t value)  
*Sets the value for the DAC interval event.*

## Pulse-Out Trigger

- static void [PDB\\_EnablePulseOutTrigger](#) (PDB\_Type \*base, [pdb\\_pulse\\_out\\_channel\\_mask\\_t](#) channelMask, bool enable)  
*Enables the pulse out trigger channels.*
- static void [PDB\\_SetPulseOutTriggerDelayValue](#) (PDB\_Type \*base, [pdb\\_pulse\\_out\\_trigger\\_channel\\_t](#) channel, uint32\_t value1, uint32\_t value2)  
*Sets event values for the pulse out trigger.*

## 23.3 Data Structure Documentation

### 23.3.1 struct [pdb\\_config\\_t](#)

#### Data Fields

- [pdb\\_load\\_value\\_mode\\_t](#) loadValueMode  
*Select the load value mode.*
- [pdb\\_prescaler\\_divider\\_t](#) prescalerDivider  
*Select the prescaler divider.*
- [pdb\\_divider\\_multiplication\\_factor\\_t](#) dividerMultiplicationFactor  
*Multiplication factor select for prescaler.*
- [pdb\\_trigger\\_input\\_source\\_t](#) triggerInputSource  
*Select the trigger input source.*
- bool enableContinuousMode  
*Enable the PDB operation in Continuous mode.*

## Data Structure Documentation

### 23.3.1.0.0.60 Field Documentation

23.3.1.0.0.60.1  `pdb_load_value_mode_t pdb_config_t::loadValueMode`

23.3.1.0.0.60.2  `pdb_prescaler_divider_t pdb_config_t::prescalerDivider`

23.3.1.0.0.60.3  `pdb_divider_multiplication_factor_t pdb_config_t::dividerMultiplicationFactor`

23.3.1.0.0.60.4  `pdb_trigger_input_source_t pdb_config_t::triggerInputSource`

23.3.1.0.0.60.5  `bool pdb_config_t::enableContinuousMode`

### 23.3.2 `struct pdb_adc_pretrigger_config_t`

#### Data Fields

- `uint32_t enablePreTriggerMask`  
*PDB Channel Pre-trigger Enable.*
- `uint32_t enableOutputMask`  
*PDB Channel Pre-trigger Output Select.*
- `uint32_t enableBackToBackOperationMask`  
*PDB Channel pre-trigger Back-to-Back Operation Enable.*

### 23.3.2.0.0.61 Field Documentation

23.3.2.0.0.61.1  `uint32_t pdb_adc_pretrigger_config_t::enablePreTriggerMask`

23.3.2.0.0.61.2  `uint32_t pdb_adc_pretrigger_config_t::enableOutputMask`

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

23.3.2.0.0.61.3  `uint32_t pdb_adc_pretrigger_config_t::enableBackToBackOperationMask`

Back-to-back operation enables the ADC conversions complete to trigger the next PDB channel pre-trigger and trigger output, so that the ADC conversions can be triggered on next set of configuration and results registers.

### 23.3.3 `struct pdb_dac_trigger_config_t`

#### Data Fields

- `bool enableExternalTriggerInput`  
*Enables the external trigger for DAC interval counter.*
- `bool enableIntervalTrigger`  
*Enables the DAC interval trigger.*

**23.3.3.0.0.62 Field Documentation****23.3.3.0.0.62.1 bool pdb\_dac\_trigger\_config\_t::enableExternalTriggerInput****23.3.3.0.0.62.2 bool pdb\_dac\_trigger\_config\_t::enableIntervalTrigger****23.4 Macro Definition Documentation****23.4.1 #define FSL\_PDB\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))****23.5 Enumeration Type Documentation****23.5.1 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.

**23.5.2 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.

**23.5.3 enum \_pdb\_interrupt\_enable**

Enumerator

*kPDB\_SequenceErrorInterruptEnable* PDB sequence error interrupt enable.

*kPDB\_DelayInterruptEnable* PDB delay interrupt enable.

**23.5.4 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)

## Enumeration Type Documentation

- 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.

### 23.5.5 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.

### 23.5.6 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.

### 23.5.7 enum pdb\_trigger\_input\_source\_t

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger. See chip configuration details for the actual PDB input trigger connections.

Enumerator

- kPDB\_TriggerInput0*** Trigger-In 0.
- kPDB\_TriggerInput1*** Trigger-In 1.
- kPDB\_TriggerInput2*** Trigger-In 2.
- kPDB\_TriggerInput3*** Trigger-In 3.
- kPDB\_TriggerInput4*** Trigger-In 4.
- kPDB\_TriggerInput5*** Trigger-In 5.
- kPDB\_TriggerInput6*** Trigger-In 6.
- kPDB\_TriggerInput7*** Trigger-In 7.
- kPDB\_TriggerInput8*** Trigger-In 8.
- kPDB\_TriggerInput9*** Trigger-In 9.
- kPDB\_TriggerInput10*** Trigger-In 10.
- kPDB\_TriggerInput11*** Trigger-In 11.
- kPDB\_TriggerInput12*** Trigger-In 12.
- kPDB\_TriggerInput13*** Trigger-In 13.
- kPDB\_TriggerInput14*** Trigger-In 14.
- kPDB\_TriggerSoftware*** Trigger-In 15, software trigger.

### 23.5.8 enum pdb\_adc\_trigger\_channel\_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kPDB\_ADCTriggerChannel0*** PDB ADC trigger channel number 0.
- kPDB\_ADCTriggerChannel1*** PDB ADC trigger channel number 1.
- kPDB\_ADCTriggerChannel2*** PDB ADC trigger channel number 2.
- kPDB\_ADCTriggerChannel3*** PDB ADC trigger channel number 3.

### 23.5.9 enum pdb\_adc\_pretrigger\_t

Note

Actual number of available pretrigger channels is SoC dependent

Enumerator

- kPDB\_ADCPreTrigger0*** PDB ADC pretrigger number 0.
- kPDB\_ADCPreTrigger1*** PDB ADC pretrigger number 1.
- kPDB\_ADCPreTrigger2*** PDB ADC pretrigger number 2.
- kPDB\_ADCPreTrigger3*** PDB ADC pretrigger number 3.

## Enumeration Type Documentation

- kPDB\_ADCPreTrigger4*** PDB ADC pretrigger number 4.
- kPDB\_ADCPreTrigger5*** PDB ADC pretrigger number 5.
- kPDB\_ADCPreTrigger6*** PDB ADC pretrigger number 6.
- kPDB\_ADCPreTrigger7*** PDB ADC pretrigger number 7.

### 23.5.10 enum pdb\_dac\_trigger\_channel\_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kPDB\_DACTriggerChannel0*** PDB DAC trigger channel number 0.
- kPDB\_DACTriggerChannel1*** PDB DAC trigger channel number 1.

### 23.5.11 enum pdb\_pulse\_out\_trigger\_channel\_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kPDB\_PulseOutTriggerChannel0*** PDB pulse out trigger channel number 0.
- kPDB\_PulseOutTriggerChannel1*** PDB pulse out trigger channel number 1.
- kPDB\_PulseOutTriggerChannel2*** PDB pulse out trigger channel number 2.
- kPDB\_PulseOutTriggerChannel3*** PDB pulse out trigger channel number 3.

### 23.5.12 enum pdb\_pulse\_out\_channel\_mask\_t

Note

Actual number of available channels mask is SoC dependent

Enumerator

- kPDB\_PulseOutChannel0Mask*** PDB pulse out trigger channel number 0 mask.
- kPDB\_PulseOutChannel1Mask*** PDB pulse out trigger channel number 1 mask.
- kPDB\_PulseOutChannel2Mask*** PDB pulse out trigger channel number 2 mask.
- kPDB\_PulseOutChannel3Mask*** PDB pulse out trigger channel number 3 mask.

## 23.6 Function Documentation

### 23.6.1 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". |

### 23.6.2 void PDB\_Deinit ( PDB\_Type \* *base* )

Parameters

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

### 23.6.3 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_PrescalerDivide1;
* config->dividerMultiplicationFactor = kPDB_DividerMultiplicationFactor1
 ;
* config->triggerInputSource = kPDB_TriggerSoftware;
* config->enableContinuousMode = false;
*
```

Parameters

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

### 23.6.4 static void PDB\_Enable ( PDB\_Type \* *base*, bool *enable* ) [inline], [static]

## Function Documentation

Parameters

|               |                              |
|---------------|------------------------------|
| <i>base</i>   | PDB peripheral base address. |
| <i>enable</i> | Enable the module or not.    |

**23.6.5 static void PDB\_DoSoftwareTrigger ( PDB\_Type \* *base* ) [inline], [static]**

Parameters

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

**23.6.6 static void PDB\_DoLoadValues ( PDB\_Type \* *base* ) [inline], [static]**

This function loads the counter values from the internal buffer. See "pdb\_load\_value\_mode\_t" about PDB's load mode.

Parameters

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

**23.6.7 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.   |

**23.6.8 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". |

### 23.6.9 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". |

### 23.6.10 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".

### 23.6.11 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". |

### 23.6.12 static void PDB\_SetModulusValue ( **PDB\_Type** \* *base*, **uint32\_t** *value* ) [**inline**], [**static**]

## Function Documentation

Parameters

|              |                                                     |
|--------------|-----------------------------------------------------|
| <i>base</i>  | PDB peripheral base address.                        |
| <i>value</i> | Setting value for the modulus. 16-bit is available. |

**23.6.13 static uint32\_t PDB\_GetCounterValue ( PDB\_Type \* *base* ) [inline], [static]**

Parameters

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

Returns

PDB counter's current value.

**23.6.14 static void PDB\_SetCounterDelayValue ( PDB\_Type \* *base*, uint32\_t *value* ) [inline], [static]**

Parameters

|              |                                                                 |
|--------------|-----------------------------------------------------------------|
| <i>base</i>  | PDB peripheral base address.                                    |
| <i>value</i> | Setting value for PDB counter delay event. 16-bit is available. |

**23.6.15 static void PDB\_SetADCPreTriggerConfig ( PDB\_Type \* *base*, pdb\_adc\_trigger\_channel\_t *channel*, pdb\_adc\_presettrigger\_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". |
|---------------|----------------------------------------------------------------------------|

**23.6.16 static void PDB\_SetADCPreTriggerDelayValue ( PDB\_Type \* *base*, pdb\_adc\_trigger\_channel\_t *channel*, pdb\_adc\_pretrigger\_t *pretriggerNumber*, 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>pretrigger-Number</i> | Channel group index for ADC instance.                               |
| <i>value</i>             | Setting value for ADC pre-trigger delay event. 16-bit is available. |

**23.6.17 static uint32\_t PDB\_GetADCPreTriggerStatusFlags ( PDB\_Type \* *base*, pdb\_adc\_trigger\_channel\_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".

**23.6.18 static void PDB\_ClearADCPreTriggerStatusFlags ( PDB\_Type \* *base*, pdb\_adc\_trigger\_channel\_t *channel*, uint32\_t *mask* ) [inline], [static]**

## Function Documentation

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". |

**23.6.19 void PDB\_SetDACTriggerConfig ( PDB\_Type \* *base*, pdb\_dac\_trigger\_channel\_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". |

**23.6.20 static void PDB\_SetDACTriggerIntervalValue ( PDB\_Type \* *base*, pdb\_dac\_trigger\_channel\_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

|                |                                           |
|----------------|-------------------------------------------|
| <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. |

**23.6.21 static void PDB\_EnablePulseOutTrigger ( PDB\_Type \* *base*, pdb\_pulse\_out\_channel\_mask\_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.                     |

### 23.6.22 static void PDB\_SetPulseOutTriggerDelayValue ( PDB\_Type \* *base*,                   pdb\_pulse\_out\_trigger\_channel\_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.             |

## Function Documentation

# Chapter 24

## PIT: Periodic Interrupt Timer

### 24.1 Overview

The MCUXpresso SDK provides a driver for the Periodic Interrupt Timer (PIT) of MCUXpresso SDK devices.

### 24.2 Function groups

The PIT driver supports operating the module as a time counter.

#### 24.2.1 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.

#### 24.2.2 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.

#### 24.2.3 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

### 24.2.4 Status

Provides functions to get and clear the PIT status.

### 24.2.5 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

## 24.3 Typical use case

### 24.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pit

## Data Structures

- struct [pit\\_config\\_t](#)  
*PIT configuration structure.* [More...](#)

## Enumerations

- enum [pit\\_chnl\\_t](#) {  
    kPIT\_Chnl\_0 = 0U,  
    kPIT\_Chnl\_1,  
    kPIT\_Chnl\_2,  
    kPIT\_Chnl\_3 }  
*List of PIT channels.*
- enum [pit\\_interrupt\\_enable\\_t](#) { kPIT\_TimerInterruptEnable = PIT\_TCTRL\_TIE\_MASK }  
*List of PIT interrupts.*
- enum [pit\\_status\\_flags\\_t](#) { kPIT\_TimerFlag = PIT\_TFLG\_TIF\_MASK }  
*List of PIT status flags.*

## Driver version

- #define [FSL\\_PIT\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 1))  
*PIT Driver Version 2.0.1.*

## 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.*

## 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.*

## 24.4 Data Structure Documentation

### 24.4.1 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*

## 24.5 Enumeration Type Documentation

### 24.5.1 enum pit\_chnl\_t

## Function Documentation

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.

### 24.5.2 enum pit\_interrupt\_enable\_t

Enumerator

- kPIT\_TimerInterruptEnable*** Timer interrupt enable.

### 24.5.3 enum pit\_status\_flags\_t

Enumerator

- kPIT\_TimerFlag*** Timer flag.

## 24.6 Function Documentation

### 24.6.1 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 |

### 24.6.2 void PIT\_Deinit ( PIT\_Type \* *base* )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | PIT peripheral base address |
|-------------|-----------------------------|

#### 24.6.3 static void PIT\_GetDefaultConfig ( pit\_config\_t \* *config* ) [inline], [static]

The default values are as follows.

```
* config->enableRunInDebug = false;
*
```

Parameters

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

#### 24.6.4 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. |

#### 24.6.5 static void PIT\_EnableInterrupts ( 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 interrupts to enable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a> |

**24.6.6 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> |

**24.6.7 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](#)

**24.6.8 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](#)

#### 24.6.9 static void PIT\_ClearStatusFlags ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|                |                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                      |
| <i>channel</i> | Timer channel number                                                                                             |
| <i>mask</i>    | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">pit_status_flags_t</a> |

#### 24.6.10 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        |

## Function Documentation

|              |                                |
|--------------|--------------------------------|
| <i>count</i> | Timer period in units of ticks |
|--------------|--------------------------------|

### 24.6.11 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

### 24.6.12 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.       |

### 24.6.13 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.       |

## Function Documentation

# Chapter 25

## PMC: Power Management Controller

### 25.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Power Management Controller (PMC) module of MCUXpresso SDK devices. The PMC module contains internal voltage regulator, power on reset, low-voltage detect system, and high-voltage detect system.

### Data Structures

- struct [pmc\\_low\\_volt\\_detect\\_config\\_t](#)  
*Low-voltage Detect Configuration Structure. [More...](#)*
- struct [pmc\\_low\\_volt\\_warning\\_config\\_t](#)  
*Low-voltage Warning Configuration Structure. [More...](#)*
- 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.*

## 25.2 Data Structure Documentation

### 25.2.1 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.*

### 25.2.2 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.*

### 25.2.3 struct pmc\_bandgap\_buffer\_config\_t

#### Data Fields

- bool [enable](#)  
*Enable bandgap buffer.*
- bool [enableInLowPowerMode](#)

*Enable bandgap buffer in low-power mode.*

### 25.2.3.0.0.63 Field Documentation

25.2.3.0.0.63.1 `bool pmc_bandgap_buffer_config_t::enable`

25.2.3.0.0.63.2 `bool pmc_bandgap_buffer_config_t::enableInLowPowerMode`

## 25.3 Macro Definition Documentation

25.3.1 `#define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

Version 2.0.0.

## 25.4 Enumeration Type Documentation

25.4.1 `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 )

25.4.2 `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)

## 25.5 Function Documentation

25.5.1 `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. |

### 25.5.2 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

### 25.5.3 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. |
|-------------|------------------------------|

### 25.5.4 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. |

### 25.5.5 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.

### 25.5.6 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. |
|-------------|------------------------------|

### 25.5.7 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 |

### 25.5.8 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.

### 25.5.9 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. |
|-------------|------------------------------|

### 25.5.10 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 26

## PORT: Port Control and Interrupts

### 26.1 Overview

The MCUXpresso SDK provides a driver for the Port Control and Interrupts (PORT) module of MCUXpresso SDK devices.

### Data Structures

- struct `port_digital_filter_config_t`  
*PORT digital filter feature configuration definition.* [More...](#)
- struct `port_pin_config_t`  
*PORT pin configuration structure.* [More...](#)

### Enumerations

- enum `_port_pull` {  
  `kPORT_PullDisable` = 0U,  
  `kPORT_PullDown` = 2U,  
  `kPORT_PullUp` = 3U }  
*Internal resistor pull feature selection.*
- enum `_port_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` {

## Overview

```
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, 1, 0))  
*Version 2.1.0.*

## Configuration

- static void `PORT_SetPinConfig` (PORT\_Type \*base, uint32\_t pin, const `port_pin_config_t` \*config)  
*Sets the port PCR register.*
- static void `PORT_SetMultiplePinsConfig` (PORT\_Type \*base, uint32\_t mask, const `port_pin_config_t` \*config)  
*Sets the port PCR register for multiple pins.*
- static void `PORT_SetPinMux` (PORT\_Type \*base, uint32\_t pin, `port_mux_t` mux)  
*Configures the pin muxing.*

- static void **PORT\_EnablePinsDigitalFilter** (PORT\_Type \*base, uint32\_t mask, bool enable)  
*Enables the digital filter in one port, each bit of the 32-bit register represents one pin.*
- static void **PORT\_SetDigitalFilterConfig** (PORT\_Type \*base, const port\_digital\_filter\_config\_t \*config)  
*Sets the digital filter in one port, each bit of the 32-bit register represents one pin.*

## Interrupt

- static void **PORT\_SetPinInterruptConfig** (PORT\_Type \*base, uint32\_t pin, port\_interrupt\_t config)  
*Configures the port pin interrupt/DMA request.*
- static void **PORT\_SetPinDriveStrength** (PORT\_Type \*base, uint32\_t pin, uint8\_t strength)  
*Configures the port pin drive strength.*
- static uint32\_t **PORT\_GetPinsInterruptFlags** (PORT\_Type \*base)  
*Reads the whole port status flag.*
- static void **PORT\_ClearPinsInterruptFlags** (PORT\_Type \*base, uint32\_t mask)  
*Clears the multiple pin interrupt status flag.*

## 26.2 Data Structure Documentation

### 26.2.1 struct port\_digital\_filter\_config\_t

#### Data Fields

- uint32\_t **digitalFilterWidth**  
*Set digital filter width.*
- port\_digital\_filter\_clock\_source\_t **clockSource**  
*Set digital filter clockSource.*

### 26.2.2 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].*

## Enumeration Type Documentation

### 26.3 Macro Definition Documentation

26.3.1 `#define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`

### 26.4 Enumeration Type Documentation

#### 26.4.1 enum \_port\_pull

Enumerator

***kPORT\_PullDisable*** Internal pull-up/down resistor is disabled.

***kPORT\_PullDown*** Internal pull-down resistor is enabled.

***kPORT\_PullUp*** Internal pull-up resistor is enabled.

#### 26.4.2 enum \_port\_slew\_rate

Enumerator

***kPORT\_FastSlewRate*** Fast slew rate is configured.

***kPORT\_SlowSlewRate*** Slow slew rate is configured.

#### 26.4.3 enum \_port\_open\_drain\_enable

Enumerator

***kPORT\_OpenDrainDisable*** Open drain output is disabled.

***kPORT\_OpenDrainEnable*** Open drain output is enabled.

#### 26.4.4 enum \_port\_passive\_filter\_enable

Enumerator

***kPORT\_PassiveFilterDisable*** Passive input filter is disabled.

***kPORT\_PassiveFilterEnable*** Passive input filter is enabled.

#### 26.4.5 enum \_port\_drive\_strength

Enumerator

***kPORT\_LowDriveStrength*** Low-drive strength is configured.

***kPORT\_HighDriveStrength*** High-drive strength is configured.

## 26.4.6 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.

## 26.4.7 enum port\_mux\_t

Enumerator

***kPORT\_PinDisabledOrAnalog*** Corresponding pin is disabled, but is used as an analog pin.

***kPORT\_MuxAsGpio*** Corresponding pin is configured as GPIO.

***kPORT\_MuxAlt2*** Chip-specific.

***kPORT\_MuxAlt3*** Chip-specific.

***kPORT\_MuxAlt4*** Chip-specific.

***kPORT\_MuxAlt5*** Chip-specific.

***kPORT\_MuxAlt6*** Chip-specific.

***kPORT\_MuxAlt7*** Chip-specific.

***kPORT\_MuxAlt8*** Chip-specific.

***kPORT\_MuxAlt9*** Chip-specific.

***kPORT\_MuxAlt10*** Chip-specific.

***kPORT\_MuxAlt11*** Chip-specific.

***kPORT\_MuxAlt12*** Chip-specific.

***kPORT\_MuxAlt13*** Chip-specific.

***kPORT\_MuxAlt14*** Chip-specific.

***kPORT\_MuxAlt15*** Chip-specific.

## 26.4.8 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.

## Function Documentation

### 26.4.9 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.

## 26.5 Function Documentation

### 26.5.1 static void PORT\_SetPinConfig ( PORT\_Type \* *base*, uint32\_t *pin*, const port\_pin\_config\_t \* *config* ) [inline], [static]

This is an example to define an input pin or output pin PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
* kPORT_PullUp,
* kPORT_FastSlewRate,
* kPORT_PassiveFilterDisable,
* kPORT_OpenDrainDisable,
* kPORT_LowDriveStrength,
* kPORT_MuxAsGpio,
* kPORT_UnLockRegister,
* };
*
```

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | PORT peripheral base pointer.              |
| <i>pin</i>    | PORT pin number.                           |
| <i>config</i> | PORT PCR register configuration structure. |

### 26.5.2 static void PORT\_SetMultiplePinsConfig ( PORT\_Type \* *base*, uint32\_t *mask*, const port\_pin\_config\_t \* *config* ) [inline], [static]

This is an example to define input pins or output pins PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
* kPORT_PullUp ,
* kPORT_PullEnable,
* kPORT_FastSlewRate,
* kPORT_PassiveFilterDisable,
* kPORT_OpenDrainDisable,
* kPORT_LowDriveStrength,
* kPORT_MuxAsGpio,
* kPORT_UnlockRegister,
* };
*
```

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | PORT peripheral base pointer.              |
| <i>mask</i>   | PORT pin number macro.                     |
| <i>config</i> | PORT PCR register configuration structure. |

### 26.5.3 static void PORT\_SetPinMux ( PORT\_Type \* *base*, uint32\_t *pin*, port\_mux\_t *mux* ) [inline], [static]

Parameters

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | PORT peripheral base pointer.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>pin</i>  | PORT pin number.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <i>mux</i>  | <p>pin muxing slot selection.</p> <ul style="list-style-type: none"> <li>• <a href="#">kPORT_PinDisabledOrAnalog</a>: Pin disabled or work in analog function.</li> <li>• <a href="#">kPORT_MuxAsGpio</a> : Set as GPIO.</li> <li>• <a href="#">kPORT_MuxAlt2</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt3</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt4</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt5</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt6</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt7</a> : chip-specific. : 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</li> </ul> |

### 26.5.4 static void PORT\_EnablePinsDigitalFilter ( PORT\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | PORT peripheral base pointer. |
|-------------|-------------------------------|

## Function Documentation

|             |                        |
|-------------|------------------------|
| <i>mask</i> | PORT pin number macro. |
|-------------|------------------------|

### 26.5.5 static void PORT\_SetDigitalFilterConfig ( PORT\_Type \* *base*, const port\_digital\_filter\_config\_t \* *config* ) [inline], [static]

Parameters

|               |                                              |
|---------------|----------------------------------------------|
| <i>base</i>   | PORT peripheral base pointer.                |
| <i>config</i> | PORT digital filter configuration structure. |

### 26.5.6 static void PORT\_SetPinInterruptConfig ( PORT\_Type \* *base*, uint32\_t *pin*, port\_interrupt\_t *config* ) [inline], [static]

Parameters

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | PORT peripheral base pointer.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>pin</i>    | PORT pin number.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>config</i> | PORT pin interrupt configuration. <ul style="list-style-type: none"><li>• <a href="#">kPORT_InterruptOrDMADisabled</a>: Interrupt/DMA request disabled.</li><li>• <a href="#">kPORT_DMARisingEdge</a> : DMA request on rising edge(if the DMA requests exit).</li><li>• <a href="#">kPORT_DMAPFallingEdge</a>: DMA request on falling edge(if the DMA requests exit).</li><li>• <a href="#">kPORT_DMAEitherEdge</a> : DMA request on either edge(if the DMA requests exit).</li><li>• <a href="#">#kPORT_FlagRisingEdge</a> : Flag sets on rising edge(if the Flag states exit).</li><li>• <a href="#">#kPORT_FlagFallingEdge</a> : Flag sets on falling edge(if the Flag states exit).</li><li>• <a href="#">#kPORT_FlagEitherEdge</a> : Flag sets on either edge(if the Flag states exit).</li><li>• <a href="#">kPORT_InterruptLogicZero</a> : Interrupt when logic zero.</li><li>• <a href="#">kPORT_InterruptRisingEdge</a> : Interrupt on rising edge.</li><li>• <a href="#">kPORT_InterruptFallingEdge</a>: Interrupt on falling edge.</li><li>• <a href="#">kPORT_InterruptEitherEdge</a> : Interrupt on either edge.</li><li>• <a href="#">kPORT_InterruptLogicOne</a> : Interrupt when logic one.</li><li>• <a href="#">#kPORT_ActiveHighTriggerOutputEnable</a> : Enable active high-trigger output (if the trigger states exit).</li><li>• <a href="#">#kPORT_ActiveLowTriggerOutputEnable</a> : Enable active low-trigger output (if the trigger states exit).</li></ul> |

26.5.7 **static void PORT\_SetPinDriveStrength ( PORT\_Type \* *base*, uint32\_t *pin*,  
uint8\_t *strength* ) [inline], [static]**

## Function Documentation

Parameters

|               |                                                                                                                                                                                                                                                       |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | PORT peripheral base pointer.                                                                                                                                                                                                                         |
| <i>pin</i>    | PORT pin number.                                                                                                                                                                                                                                      |
| <i>config</i> | PORT pin drive strength <ul style="list-style-type: none"><li>• <a href="#">kPORT_LowDriveStrength</a> = 0U - Low-drive strength is configured.</li><li>• <a href="#">kPORT_HighDriveStrength</a> = 1U - High-drive strength is configured.</li></ul> |

### 26.5.8 static uint32\_t PORT\_GetPinsInterruptFlags ( PORT\_Type \* *base* ) [inline], [static]

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | PORT peripheral base pointer. |
|-------------|-------------------------------|

Returns

Current port interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

### 26.5.9 static void PORT\_ClearPinsInterruptFlags ( PORT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | PORT peripheral base pointer. |
| <i>mask</i> | PORT pin number macro.        |

# Chapter 27

## RCM: Reset Control Module Driver

### 27.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Reset Control Module (RCM) module of MCUXpresso SDK devices.

### Data Structures

- struct `rcm_reset_pin_filter_config_t`  
*Reset pin filter configuration. [More...](#)*

### Enumerations

- enum `rcm_reset_source_t`{  
  `kRCM_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)

## Enumeration Type Documentation

- Gets the reset source status which caused a previous reset.  
void [RCM\\_ConfigureResetPinFilter](#) (RCM\_Type \*base, const [rcm\\_reset\\_pin\\_filter\\_config\\_t](#) \*config)  
*Configures the reset pin filter.*
- static bool [RCM\\_GetEasyPortModePinStatus](#) (RCM\_Type \*base)  
*Gets the EZP\_MS\_B pin assert status.*

## 27.2 Data Structure Documentation

### 27.2.1 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.*

#### 27.2.1.0.0.64 Field Documentation

##### 27.2.1.0.0.64.1 bool rcm\_reset\_pin\_filter\_config\_t::enableFilterInStop

##### 27.2.1.0.0.64.2 rcm\_run\_wait\_filter\_mode\_t rcm\_reset\_pin\_filter\_config\_t::filterInRunWait

##### 27.2.1.0.0.64.3 uint8\_t rcm\_reset\_pin\_filter\_config\_t::busClockFilterCount

## 27.3 Macro Definition Documentation

### 27.3.1 #define FSL\_RCM\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

## 27.4 Enumeration Type Documentation

### 27.4.1 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.

## 27.4.2 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.

## 27.5 Function Documentation

### 27.5.1 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.

### 27.5.2 void RCM\_ConfigureResetPinFilter ( RCM\_Type \* *base*, const rcm\_reset\_pin\_filter\_config\_t \* *config* )

This function sets the reset pin filter including the filter source, filter width, and so on.

## Function Documentation

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | RCM peripheral base address.            |
| <i>config</i> | Pointer to the configuration structure. |

### 27.5.3 **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

# Chapter 28

## RNGA: Random Number Generator Accelerator Driver

### 28.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Random Number Generator Accelerator (RNGA) block of MCUXpresso SDK devices.

### 28.2 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.

### 28.3 Get random data from RNGA

1. [RNGA\\_GetRandomData\(\)](#) function gets random data from the RNGA module.

### 28.4 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.

### 28.5 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:

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

#### 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.

## Macro Definition Documentation

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.*

## 28.6 Macro Definition Documentation

### 28.6.1 #define FSL\_RNGA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

## 28.7 Enumeration Type Documentation

### 28.7.1 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.

## 28.8 Function Documentation

### 28.8.1 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 |
|-------------|-------------------|

### 28.8.2 void RNGA\_Deinit ( RNG\_Type \* *base* )

This function shuts down the RNGA.

Parameters

|             |                   |
|-------------|-------------------|
| <i>base</i> | RNGA base address |
|-------------|-------------------|

### 28.8.3 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 |

## Function Documentation

|                  |                       |
|------------------|-----------------------|
| <i>data_size</i> | size of data in bytes |
|------------------|-----------------------|

Returns

RNGA status

### 28.8.4 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  |

### 28.8.5 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 |

### 28.8.6 rnga\_mode\_t RNGA\_GetMode ( RNG\_Type \* *base* )

This function gets the RNGA working mode.

Parameters

|             |                   |
|-------------|-------------------|
| <i>base</i> | RNGA base address |
|-------------|-------------------|

Returns

normal mode or sleep mode

# Chapter 29

## RTC: Real Time Clock

### 29.1 Overview

The MCUXpresso SDK provides a driver for the Real Time Clock (RTC) of MCUXpresso SDK devices.

### 29.2 Function groups

The RTC driver supports operating the module as a time counter.

#### 29.2.1 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.

#### 29.2.2 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.

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

#### 29.2.3 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.

#### 29.2.4 Start & Stop timer

The function [RTC\\_StartTimer\(\)](#) starts the RTC time counter.

The function [RTC\\_StopTimer\(\)](#) stops the RTC time counter.

## Typical use case

### 29.2.5 Status

Provides functions to get and clear the RTC status.

### 29.2.6 Interrupt

Provides functions to enable/disable RTC interrupts and get current enabled interrupts.

### 29.2.7 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.

### 29.2.8 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.

## 29.3 Typical use case

### 29.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/rtc

## 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 = (1U << 0U),  
  kRTC\_TimeOverflowInterruptEnable = (1U << 1U),  
  kRTC\_AlarmInterruptEnable = (1U << 2U),

- ```

kRTC_SecondsInterruptEnable = (1U << 3U) }

List of RTC interrupts.
• enum rtc_status_flags_t {
    kRTC_TimeInvalidFlag = (1U << 0U),
    kRTC_TimeOverflowFlag = (1U << 1U),
    kRTC_AlarmFlag = (1U << 2U) }

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_SetClockSource** (RTC_Type *base)
 Set RTC clock source.
- 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, 1, 0))
 Version 2.1.0.

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.

Data Structure Documentation

Interrupt Interface

- void [RTC_EnableInterrupts](#) (RTC_Type *base, uint32_t mask)
Enables the selected RTC interrupts.
- void [RTC_DisableInterrupts](#) (RTC_Type *base, uint32_t mask)
Disables the selected RTC interrupts.
- uint32_t [RTC_GetEnabledInterrupts](#) (RTC_Type *base)
Gets the enabled RTC interrupts.

Status Interface

- 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.

29.4 Data Structure Documentation

29.4.1 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.

29.4.1.0.0.65 Field Documentation

- 29.4.1.0.0.65.1 `uint16_t rtc_datetime_t::year`
- 29.4.1.0.0.65.2 `uint8_t rtc_datetime_t::month`
- 29.4.1.0.0.65.3 `uint8_t rtc_datetime_t::day`
- 29.4.1.0.0.65.4 `uint8_t rtc_datetime_t::hour`
- 29.4.1.0.0.65.5 `uint8_t rtc_datetime_t::minute`
- 29.4.1.0.0.65.6 `uint8_t rtc_datetime_t::second`

29.4.2 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.

29.5 Enumeration Type Documentation

29.5.1 enum rtc_interrupt_enable_t

Enumerator

- `kRTC_TimeInvalidInterruptEnable`** Time invalid interrupt.
- `kRTC_TimeOverflowInterruptEnable`** Time overflow interrupt.
- `kRTC_AlarmInterruptEnable`** Alarm interrupt.
- `kRTC_SecondsInterruptEnable`** Seconds interrupt.

Function Documentation

29.5.2 enum rtc_status_flags_t

Enumerator

kRTC_TimeInvalidFlag Time invalid flag.

kRTC_TimeOverflowFlag Time overflow flag.

kRTC_AlarmFlag Alarm flag.

29.5.3 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

29.6 Function Documentation

29.6.1 void RTC_Init (RTC_Type * *base*, const rtc_config_t * *config*)

This function issues a software reset if the timer invalid flag is set.

Note

This API should be called at the beginning of the application using the RTC driver.

Parameters

<i>base</i>	RTC peripheral base address
<i>config</i>	Pointer to the user's RTC configuration structure.

29.6.2 static void RTC_Deinit (RTC_Type * *base*) [inline], [static]

Parameters

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

29.6.3 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.
---------------	--

29.6.4 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

29.6.5 void RTC_GetDatetime (*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 date and time details are stored.

29.6.6 **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

29.6.7 **void RTC_GetAlarm (RTC_Type * *base*, rtc_datetime_t * *datetime*)**

Parameters

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to the structure where the alarm date and time details are stored.

29.6.8 **void RTC_EnableInterrupts (RTC_Type * *base*, uint32_t *mask*)**

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
-------------	---

29.6.9 void RTC_DisableInterrupts (RTC_Type * *base*, uint32_t *mask*)

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

29.6.10 uint32_t RTC_GetEnabledInterrupts (RTC_Type * *base*)

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](#)**29.6.11 uint32_t RTC_GetStatusFlags (RTC_Type * *base*)**

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](#)**29.6.12 void RTC_ClearStatusFlags (RTC_Type * *base*, uint32_t *mask*)**

Function Documentation

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

29.6.13 static void RTC_SetClockSource(RTC_Type * *base*) [inline], [static]

Parameters

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

Note

After setting this bit, wait the oscillator startup time before enabling the time counter to allow the 32.768 kHz clock time to stabilize.

29.6.14 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
-------------	-----------------------------

29.6.15 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
-------------	-----------------------------

29.6.16 static void RTC_SetOscCapLoad(RTC_Type * *base*, uint32_t *capLoad*) [inline], [static]

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

29.6.17 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
-------------	-----------------------------

Function Documentation

Chapter 30

SAI: Serial Audio Interface

30.1 Overview

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

SAI driver includes functional APIs and transactional APIs.

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

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the `sai_handle_t` as the first parameter. Initialize the handle by calling the [SAI_TransferTxCreateHandle\(\)](#) or [SAI_TransferRxCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer. This means that the functions [SAI_TransferSendNonBlocking\(\)](#) and [SAI_TransfsterReceiveNonBlocking\(\)](#) 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.

30.2 Typical use case

30.2.1 SAI Send/receive using an interrupt method

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

30.2.2 SAI Send/receive using a DMA method

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

Modules

- [SAI DMA Driver](#)
- [SAI EDMA Driver](#)
- [SAI SDMA Driver](#)

Typical use case

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.
- enum [_sai_channel_mask](#) {
 kSAI_Channel0Mask = 1 << 0U,
 kSAI_Channel1Mask = 1 << 1U,
 kSAI_Channel2Mask = 1 << 2U,
 kSAI_Channel3Mask = 1 << 3U,
 kSAI_Channel4Mask = 1 << 4U,
 kSAI_Channel5Mask = 1 << 5U,
 kSAI_Channel6Mask = 1 << 6U,
 kSAI_Channel7Mask = 1 << 7U }
- 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_MonoRight`,

 `kSAI_MonoLeft` }

 Mono or stereo audio format.
- enum `sai_data_order_t` {

 `kSAI_DataLSB` = 0x0U,

 `kSAI_DataMSB` }

 SAI data order, MSB or LSB.
- enum `sai_clock_polarity_t` {

 `kSAI_PolarityActiveHigh` = 0x0U,

 `kSAI_PolarityActiveLow` }

 SAI clock polarity, active high or low.
- enum `sai_sync_mode_t` {

 `kSAI_ModeAsync` = 0x0U,

 `kSAI_ModeSync`,

 `kSAI_ModeSyncWithOtherTx`,

 `kSAI_ModeSyncWithOtherRx` }

 Synchronous or asynchronous mode.
- enum `sai_mclk_source_t` {

 `kSAI_MclkSourceSysclk` = 0x0U,

 `kSAI_MclkSourceSelect1`,

 `kSAI_MclkSourceSelect2`,

 `kSAI_MclkSourceSelect3` }

 Master clock source.
- enum `sai_bclk_source_t` {

 `kSAI_BclkSourceBusclk` = 0x0U,

 `kSAI_BclkSourceMclkOption1` = 0x1U,

 `kSAI_BclkSourceMclkOption2` = 0x2U,

 `kSAI_BclkSourceMclkOption3` = 0x3U,

 `kSAI_BclkSourceMclkDiv` = 0x1U,

 `kSAI_BclkSourceOtherSai0` = 0x2U,

 `kSAI_BclkSourceOtherSai1` = 0x3U }
- 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` {

Typical use case

```
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_sample_rate_t` {
 kSAI_SampleRate8KHz = 8000U,
 kSAI_SampleRate11025Hz = 11025U,
 kSAI_SampleRate12KHz = 12000U,
 kSAI_SampleRate16KHz = 16000U,
 kSAI_SampleRate22050Hz = 22050U,
 kSAI_SampleRate24KHz = 24000U,
 kSAI_SampleRate32KHz = 32000U,
 kSAI_SampleRate44100Hz = 44100U,
 kSAI_SampleRate48KHz = 48000U,
 kSAI_SampleRate96KHz = 96000U,
 kSAI_SampleRate192KHz = 192000U,
 kSAI_SampleRate384KHz = 384000U }

Audio sample rate.

- enum `sai_word_width_t` {
 kSAI_WordWidth8bits = 8U,
 kSAI_WordWidth16bits = 16U,
 kSAI_WordWidth24bits = 24U,
 kSAI_WordWidth32bits = 32U }

Audio word width.

Driver version

- #define `FSL_SAI_DRIVER_VERSION` (MAKE_VERSION(2, 1, 7))

Version 2.1.7.

Initialization and deinitialization

- void `SAI_TxInit` (I2S_Type *base, const `sai_config_t` *config)
Initializes the SAI Tx peripheral.
- void `SAI_RxInit` (I2S_Type *base, const `sai_config_t` *config)
Initializes the SAI Rx peripheral.

- void **SAI_TxGetDefaultConfig** (sai_config_t *config)
Sets the SAI Tx configuration structure to default values.
- void **SAI_RxGetDefaultConfig** (sai_config_t *config)
Sets the SAI Rx configuration structure to default values.
- void **SAI_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.
- void **SAI_TxSoftwareReset** (I2S_Type *base, sai_reset_type_t type)
Do software reset or FIFO reset .
- void **SAI_RxSoftwareReset** (I2S_Type *base, sai_reset_type_t type)
Do software reset or FIFO reset .
- void **SAI_TxSetChannelFIFOMask** (I2S_Type *base, uint8_t mask)
Set the Tx channel FIFO enable mask.
- void **SAI_RxSetChannelFIFOMask** (I2S_Type *base, uint8_t mask)
Set the Rx channel FIFO enable mask.
- void **SAI_TxSetDataOrder** (I2S_Type *base, sai_data_order_t order)
Set the Tx data order.
- void **SAI_RxSetDataOrder** (I2S_Type *base, sai_data_order_t order)
Set the Rx data order.
- void **SAI_TxSetBitClockPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Tx data order.
- void **SAI_RxSetBitClockPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Rx data order.
- void **SAI_TxSetFrameSyncPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Tx data order.
- void **SAI_RxSetFrameSyncPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Rx data order.

Interrupts

- static void **SAI_TxEnableInterrupts** (I2S_Type *base, uint32_t mask)
Enables the SAI Tx interrupt requests.
- static void **SAI_RxEnableInterrupts** (I2S_Type *base, uint32_t mask)

Typical use case

- 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.
- void **SAI_WriteMultiChannelBlocking** (I2S_Type *base, uint32_t channel, uint32_t channelMask, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Sends data to multi channel using a blocking method.
- static void **SAI_WriteData** (I2S_Type *base, uint32_t channel, uint32_t data)
Writes data into SAI FIFO.
- void **SAI_ReadBlocking** (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Receives data using a blocking method.
- void **SAI_ReadMultiChannelBlocking** (I2S_Type *base, uint32_t channel, uint32_t channelMask, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Receives multi channel data using a blocking method.
- static uint32_t **SAI_ReadData** (I2S_Type *base, uint32_t channel)
Reads data from the SAI FIFO.

Transactional

- void **SAI_TransferTxCreateHandle** (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
Initializes the SAI Tx handle.
- void **SAI_TransferRxCreateHandle** (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
Initializes the SAI Rx handle.

- status_t **SAI_TransferTxSetFormat** (I2S_Type *base, sai_handle_t *handle, **sai_transfer_format_t** *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- status_t **SAI_TransferRxSetFormat** (I2S_Type *base, sai_handle_t *handle, **sai_transfer_format_t** *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- status_t **SAI_TransferSendNonBlocking** (I2S_Type *base, sai_handle_t *handle, **sai_transfer_t** *xfer)
Performs an interrupt non-blocking send transfer on SAI.
- status_t **SAI_TransferReceiveNonBlocking** (I2S_Type *base, sai_handle_t *handle, **sai_transfer_t** *xfer)
Performs an interrupt non-blocking receive transfer on SAI.
- status_t **SAI_TransferGetSendCount** (I2S_Type *base, sai_handle_t *handle, size_t *count)
Gets a set byte count.
- status_t **SAI_TransferGetReceiveCount** (I2S_Type *base, sai_handle_t *handle, size_t *count)
Gets a received byte count.
- void **SAI_TransferAbortSend** (I2S_Type *base, sai_handle_t *handle)
Aborts the current send.
- void **SAI_TransferAbortReceive** (I2S_Type *base, sai_handle_t *handle)
Aborts the current IRQ receive.
- void **SAI_TransferTerminateSend** (I2S_Type *base, sai_handle_t *handle)
Terminate all SAI send.
- void **SAI_TransferTerminateReceive** (I2S_Type *base, sai_handle_t *handle)
Terminate all SAI receive.
- void **SAI_TransferTxHandleIRQ** (I2S_Type *base, sai_handle_t *handle)
Tx interrupt handler.
- void **SAI_TransferRxHandleIRQ** (I2S_Type *base, sai_handle_t *handle)
Rx interrupt handler.

30.3 Data Structure Documentation

30.3.1 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.

Data Structure Documentation

30.3.2 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`
Transfer start channel.
- `uint8_t channelMask`
enabled channel mask value, reference _sai_channel_mask
- `uint8_t endChannel`
end channel number
- `uint8_t channelNums`
Total enabled channel numbers.
- `sai_protocol_t protocol`
Which audio protocol used.
- `bool isFrameSyncCompact`
True means Frame sync length is configurable according to bitWidth, false means frame sync length is 64 times of bit clock.

30.3.2.0.0.66 Field Documentation

30.3.2.0.0.66.1 bool sai_transfer_format_t::isFrameSyncCompact

30.3.3 struct sai_transfer_t

Data Fields

- `uint8_t * data`
Data start address to transfer.
- `size_t dataSize`
Transfer size.

30.3.3.0.0.67 Field Documentation**30.3.3.0.0.67.1 uint8_t* sai_transfer_t::data****30.3.3.0.0.67.2 size_t sai_transfer_t::dataSize****30.3.4 struct _sai_handle****Data Fields**

- I2S_Type * **base**
base address
- uint32_t **state**
Transfer status.
- sai_transfer_callback_t **callback**
Callback function called at transfer event.
- void * **userData**
Callback parameter passed to callback function.
- uint8_t **bitWidth**
Bit width for transfer, 8/16/24/32 bits.
- uint8_t **channel**
Transfer start channel.
- uint8_t **channelMask**
enabled channel mask value, refernece _sai_channel_mask
- uint8_t **endChannel**
end channel number
- uint8_t **channelNums**
Total enabled channel numbers.
- sai_transfer_t **saiQueue [SAI_XFER_QUEUE_SIZE]**
Transfer queue storing queued transfer.
- size_t **transferSize [SAI_XFER_QUEUE_SIZE]**
Data bytes need to transfer.
- volatile uint8_t **queueUser**
Index for user to queue transfer.
- volatile uint8_t **queueDriver**
Index for driver to get the transfer data and size.
- uint8_t **watermark**
Watermark value.

30.4 Macro Definition Documentation**30.4.1 #define SAI_XFER_QUEUE_SIZE (4)****30.5 Enumeration Type Documentation****30.5.1 enum _sai_status_t**

Enumerator

kStatus_SAI_TxBusy SAI Tx is busy.

Enumeration Type Documentation

kStatus_SAI_RxBusy SAI Rx is busy.
kStatus_SAI_TxError SAI Tx FIFO error.
kStatus_SAI_RxError SAI Rx FIFO error.
kStatus_SAI_QueueFull SAI transfer queue is full.
kStatus_SAI_TxIdle SAI Tx is idle.
kStatus_SAI_RxIdle SAI Rx is idle.

30.5.2 enum _sai_channel_mask

Enumerator

kSAI_Channel0Mask channel 0 mask value
kSAI_Channel1Mask channel 1 mask value
kSAI_Channel2Mask channel 2 mask value
kSAI_Channel3Mask channel 3 mask value
kSAI_Channel4Mask channel 4 mask value
kSAI_Channel5Mask channel 5 mask value
kSAI_Channel6Mask channel 6 mask value
kSAI_Channel7Mask channel 7 mask value

30.5.3 enum sai_protocol_t

Enumerator

kSAI_BusLeftJustified Uses left justified format.
kSAI_BusRightJustified Uses right justified format.
kSAI_BusI2S Uses I2S format.
kSAI_BusPCMA Uses I2S PCM A format.
kSAI_BusPCMB Uses I2S PCM B format.

30.5.4 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode.
kSAI_Slave Slave mode.

30.5.5 enum sai_mono_stereo_t

Enumerator

kSAI_Stereo Stereo sound.

kSAI_MonoRight Only Right channel have sound.

kSAI_MonoLeft Only left channel have sound.

30.5.6 enum sai_data_order_t

Enumerator

kSAI_DataLSB LSB bit transferred first.

kSAI_DataMSB MSB bit transferred first.

30.5.7 enum sai_clock_polarity_t

Enumerator

kSAI_PolarityActiveHigh Clock active high.

kSAI_PolarityActiveLow Clock active low.

30.5.8 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.

30.5.9 enum sai_mclk_source_t

Enumerator

kSAI_MclkSourceSysclk Master clock from the system clock.

kSAI_MclkSourceSelect1 Master clock from source 1.

kSAI_MclkSourceSelect2 Master clock from source 2.

kSAI_MclkSourceSelect3 Master clock from source 3.

Enumeration Type Documentation

30.5.10 enum sai_bclk_source_t

Enumerator

- kSAI_BclkSourceBusclk* Bit clock using bus clock.
- kSAI_BclkSourceMclkOption1* Bit clock MCLK option 1.
- kSAI_BclkSourceMclkOption2* Bit clock MCLK option2.
- kSAI_BclkSourceMclkOption3* Bit clock MCLK option3.
- kSAI_BclkSourceMclkDiv* Bit clock using master clock divider.
- kSAI_BclkSourceOtherSai0* Bit clock from other SAI device.
- kSAI_BclkSourceOtherSai1* Bit clock from other SAI device.

30.5.11 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_FIFOResponseInterruptEnable* FIFO request, means reached watermark.

30.5.12 enum _sai_dma_enable_t

Enumerator

- kSAI_FIFOWarningDMAEnable* FIFO warning caused by the DMA request.
- kSAI_FIFOResponseDMAEnable* FIFO request caused by the DMA request.

30.5.13 enum _sai_flags

Enumerator

- kSAI_WordStartFlag* Word start flag, means the first word in a frame detected.
- kSAI_SyncErrorFlag* Sync error flag, means the sync error is detected.
- kSAI_FIFOErrorFlag* FIFO error flag.
- kSAI_FIFOResponseFlag* FIFO request flag.
- kSAI_FIFOWarningFlag* FIFO warning flag.

30.5.14 enum sai_reset_type_t

Enumerator

kSAI_ResetTypeSoftware Software reset, reset the logic state.

kSAI_ResetTypeFIFO FIFO reset, reset the FIFO read and write pointer.

kSAI_ResetAll All reset.

30.5.15 enum sai_sample_rate_t

Enumerator

kSAI_SampleRate8KHz Sample rate 8000 Hz.

kSAI_SampleRate11025KHz Sample rate 11025 Hz.

kSAI_SampleRate12KHz Sample rate 12000 Hz.

kSAI_SampleRate16KHz Sample rate 16000 Hz.

kSAI_SampleRate22050KHz Sample rate 22050 Hz.

kSAI_SampleRate24KHz Sample rate 24000 Hz.

kSAI_SampleRate32KHz Sample rate 32000 Hz.

kSAI_SampleRate44100KHz Sample rate 44100 Hz.

kSAI_SampleRate48KHz Sample rate 48000 Hz.

kSAI_SampleRate96KHz Sample rate 96000 Hz.

kSAI_SampleRate192KHz Sample rate 192000 Hz.

kSAI_SampleRate384KHz Sample rate 384000 Hz.

30.5.16 enum sai_word_width_t

Enumerator

kSAI_WordWidth8bits Audio data width 8 bits.

kSAI_WordWidth16bits Audio data width 16 bits.

kSAI_WordWidth24bits Audio data width 24 bits.

kSAI_WordWidth32bits Audio data width 32 bits.

30.6 Function Documentation

30.6.1 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.

Function Documentation

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

30.6.2 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.

30.6.3 void SAI_TxGetDefaultConfig (sai_config_t * *config*)

This API initializes the configuration structure for use in SAI_TxConfig(). The initialized structure can remain unchanged in SAI_TxConfig(), or it can be modified before calling SAI_TxConfig(). This is an example.

```
sai_config_t config;  
SAI_TxGetDefaultConfig(&config);
```

Parameters

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

30.6.4 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
---------------	---

30.6.5 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
-------------	------------------

30.6.6 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
-------------	------------------

30.6.7 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
-------------	------------------

30.6.8 void SAI_TxEnable (I2S_Type * *base*, bool *enable*)

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Function Documentation

<i>enable</i>	True means enable SAI Tx, false means disable.
---------------	--

30.6.9 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.

30.6.10 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.

30.6.11 static void SAI_TxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	State mask. It can be a combination of the following source if defined: <ul style="list-style-type: none">• kSAI_WordStartFlag• kSAI_SyncErrorFlag• kSAI_FIFOErrorFlag

30.6.12 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.

30.6.13 static void SAI_RxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>State mask. It can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

30.6.14 void SAI_TxSoftwareReset (I2S_Type * *base*, sai_reset_type_t *type*)

FIFO reset means clear all the data in the FIFO, and make the FIFO pointer both to 0. Software reset means clear the Tx internal logic, including the bit clock, frame count etc. But software reset will not clear any configuration registers like TCR1~TCR5. This function will also clear all the error flags such as FIFO error, sync error etc.

Parameters

<i>base</i>	SAI base pointer
<i>type</i>	Reset type, FIFO reset or software reset

30.6.15 void SAI_RxSoftwareReset (I2S_Type * *base*, sai_reset_type_t *type*)

FIFO reset means clear all the data in the FIFO, and make the FIFO pointer both to 0. Software reset means clear the Rx internal logic, including the bit clock, frame count etc. But software reset will not clear any configuration registers like RCR1~RCR5. This function will also clear all the error flags such as FIFO error, sync error etc.

Function Documentation

Parameters

<i>base</i>	SAI base pointer
<i>type</i>	Reset type, FIFO reset or software reset

30.6.16 void SAI_TxSetChannelFIFOMask (I2S_Type * *base*, uint8_t *mask*)

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	Channel enable mask, 0 means all channel FIFO disabled, 1 means channel 0 enabled, 3 means both channel 0 and channel 1 enabled.

30.6.17 void SAI_RxSetChannelFIFOMask (I2S_Type * *base*, uint8_t *mask*)

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	Channel enable mask, 0 means all channel FIFO disabled, 1 means channel 0 enabled, 3 means both channel 0 and channel 1 enabled.

30.6.18 void SAI_TxSetDataOrder (I2S_Type * *base*, sai_data_order_t *order*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

30.6.19 void SAI_RxSetDataOrder (I2S_Type * *base*, sai_data_order_t *order*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

30.6.20 void SAI_TxSetBitClockPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

30.6.21 void SAI_RxSetBitClockPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

30.6.22 void SAI_TxSetFrameSyncPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

30.6.23 void SAI_RxSetFrameSyncPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Function Documentation

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

**30.6.24 static void SAI_TxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_WordStartInterruptEnable• kSAI_SyncErrorInterruptEnable• kSAI_FIFOWarningInterruptEnable• kSAI_FIFORequestInterruptEnable• kSAI_FIFOErrorInterruptEnable

**30.6.25 static void SAI_RxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_WordStartInterruptEnable• kSAI_SyncErrorInterruptEnable• kSAI_FIFOWarningInterruptEnable• kSAI_FIFORequestInterruptEnable• kSAI_FIFOErrorInterruptEnable

**30.6.26 static void SAI_TxDisableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

30.6.27 static void SAI_RxDisableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

30.6.28 static void SAI_TxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>DMA source The parameter can be combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

Function Documentation

30.6.29 **static void SAI_RxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	DMA source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"> • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

30.6.30 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.

30.6.31 static uint32_t SAI_RxGetDataRegisterAddress (I2S_Type * *base*, uint32_t *channel*) [inline], [static]

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

Function Documentation

30.6.32 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.

30.6.33 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.

30.6.34 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.

Function Documentation

30.6.35 void SAI_WriteMultiChannelBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *channelMask*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>channelMask</i>	channel mask.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be written.
<i>size</i>	Bytes to be written.

30.6.36 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.

30.6.37 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.

30.6.38 void SAI_ReadMultiChannelBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *channelMask*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>channelMask</i>	channel mask.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be read.
<i>size</i>	Bytes to be read.

30.6.39 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.

30.6.40 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.

Function Documentation

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

30.6.41 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.

30.6.42 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.

30.6.43 status_t SAI_TransferRxSetFormat (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is one of status_t.

30.6.44 status_t SAI_TransferSendNonBlocking (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>xfer</i>	Pointer to the sai_transfer_t structure.

Function Documentation

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.

30.6.45 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 <i>sai_handle_t</i> 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.

30.6.46 status_t SAI_TransferGetSendCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the <i>sai_handle_t</i> structure which stores the transfer state.
<i>count</i>	Bytes count sent.

Return values

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

30.6.47 **status_t SAI_TransferGetReceiveCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count received.

Return values

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

30.6.48 **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.

30.6.49 **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.

Function Documentation

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

30.6.50 void SAI_TransferTerminateSend (I2S_Type * *base*, sai_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortSend.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

30.6.51 void SAI_TransferTerminateReceive (I2S_Type * *base*, sai_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceive.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

30.6.52 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.

30.6.53 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.

30.7 SAI DMA Driver

30.7.1 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.

Driver version

- `#define FSL_SAI_DMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 5))`
Version 2.1.5.

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.

30.7.2 Data Structure Documentation

30.7.2.1 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.

30.7.2.1.0.68 Field Documentation

30.7.2.1.0.68.1 [sai_transfer_t](#) **sai_dma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]**

30.7.2.1.0.68.2 **volatile uint8_t** **sai_dma_handle_t::queueUser**

30.7.3 Function Documentation

30.7.3.1 **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.

SAI DMA Driver

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.

30.7.3.2 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.

30.7.3.3 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.

30.7.3.4 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.

30.7.3.5 status_t SAI_TransferSendDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_transfer_t * *xfer*)

SAI DMA Driver

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.

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.

30.7.3.6 **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.

30.7.3.7 **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.

30.7.3.8 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.

30.7.3.9 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.

30.7.3.10 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.

30.8 SAI EDMA Driver

30.8.1 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.

Driver version

- #define [FSL_SAI_EDMA_DRIVER_VERSION](#) (MAKE_VERSION(2, 1, 5))
Version 2.1.5.

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_TransferTerminateSendEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle)
Terminate all SAI send.
- void [SAI_TransferTerminateReceiveEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle)
Terminate all SAI receive.
- void [SAI_TransferAbortSendEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle)

SAI EDMA Driver

- 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.

30.8.2 Data Structure Documentation

30.8.2.1 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.
- **uint8_t tcd [(SAI_XFER_QUEUE_SIZE+1U)*sizeof(edma_tcd_t)]**
 TCD pool for eDMA transfer.
- **sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]**
 Transfer queue storing queued transfer.
- **size_t transferSize [SAI_XFER_QUEUE_SIZE]**
 Data bytes need to transfer.
- **volatile uint8_t queueUser**
 Index for user to queue transfer.
- **volatile uint8_t queueDriver**
 Index for driver to get the transfer data and size.

30.8.2.1.0.69 Field Documentation

30.8.2.1.0.69.1 `uint8_t sai_edma_handle_t::nbytes`

30.8.2.1.0.69.2 `uint8_t sai_edma_handle_t::tcd[(SAI_XFER_QUEUE_SIZE+1U)*sizeof(edma_tcd_t)]`

30.8.2.1.0.69.3 `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

30.8.2.1.0.69.4 `volatile uint8_t sai_edma_handle_t::queueUser`

30.8.3 Function Documentation

30.8.3.1 `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.

30.8.3.2 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.

30.8.3.3 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.

30.8.3.4 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.

30.8.3.5 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.

30.8.3.6 **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.

30.8.3.7 **void SAI_TransferTerminateSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)**

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortSendEDMA.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

30.8.3.8 void SAI_TransferTerminateReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceiveEDMA.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

30.8.3.9 void SAI_TransferAbortSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call SAI_TransferTerminateSendEDMA.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

30.8.3.10 void SAI_TransferAbortReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call SAI_TransferTerminateReceiveEDMA.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

SAI EDMA Driver

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

30.8.3.11 **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.

30.8.3.12 **status_t SAI_TransferGetReceiveCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)**

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is no non-blocking transaction in progress.

30.9 SAI SDMA Driver

30.9.1 Overview

Data Structures

- struct `sai_sdma_handle_t`
SAI DMA transfer handle, users should not touch the content of the handle. [More...](#)

Typedefs

- typedef void(* `sai_sdma_callback_t`)(I2S_Type *base, sai_sdma_handle_t *handle, status_t status, void *userData)
SAI SDMA transfer callback function for finish and error.

Driver version

- #define `FSL_SAI_SDMA_DRIVER_VERSION` (MAKE_VERSION(2, 1, 5))
Version 2.1.5.

SDMA Transactional

- void `SAI_TransferTxCreateHandleSDMA` (I2S_Type *base, sai_sdma_handle_t *handle, `sai_sdma_callback_t` callback, void *userData, sdma_handle_t *dmaHandle, uint32_t eventSource)
Initializes the SAI SDMA handle.
- void `SAI_TransferRxCreateHandleSDMA` (I2S_Type *base, sai_sdma_handle_t *handle, `sai_sdma_callback_t` callback, void *userData, sdma_handle_t *dmaHandle, uint32_t eventSource)
Initializes the SAI Rx SDMA handle.
- void `SAI_TransferTxSetFormatSDMA` (I2S_Type *base, sai_sdma_handle_t *handle, `sai_transfer_format_t` *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- void `SAI_TransferRxSetFormatSDMA` (I2S_Type *base, sai_sdma_handle_t *handle, `sai_transfer_format_t` *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- status_t `SAI_TransferSendSDMA` (I2S_Type *base, sai_sdma_handle_t *handle, `sai_transfer_t` *xfer)
Performs a non-blocking SAI transfer using DMA.
- status_t `SAI_TransferReceiveSDMA` (I2S_Type *base, sai_sdma_handle_t *handle, `sai_transfer_t` *xfer)
Performs a non-blocking SAI receive using SDMA.
- void `SAI_TransferAbortSendSDMA` (I2S_Type *base, sai_sdma_handle_t *handle)
Aborts a SAI transfer using SDMA.
- void `SAI_TransferAbortReceiveSDMA` (I2S_Type *base, sai_sdma_handle_t *handle)
Aborts a SAI receive using SDMA.

30.9.2 Data Structure Documentation

30.9.2.1 struct _sai_sdma_handle

Data Fields

- `sdma_handle_t * dmaHandle`
DMA handler for SAI send.
- `uint8_t bytesPerFrame`
Bytes in a frame.
- `uint8_t channel`
start data channel
- `uint8_t channelNums`
total transfer channel numbers, used for multififo
- `uint8_t channelMask`
enabled channel mask value, refernece _sai_channel_mask
- `uint8_t fifoOffset`
fifo address offset between multififo
- `uint8_t count`
The transfer data count in a DMA request.
- `uint32_t state`
Internal state for SAI SDMA transfer.
- `uint32_t eventSource`
SAI event source number.
- `sai_sdma_callback_t callback`
Callback for users while transfer finish or error occurs.
- `void * userData`
User callback parameter.
- `sdma_buffer_descriptor_t bdPool [SAI_XFER_QUEUE_SIZE]`
BD pool for SDMA 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.

30.9.2.1.0.70 Field Documentation

30.9.2.1.0.70.1 `sdma_buffer_descriptor_t sai_sdma_handle_t::bdPool[SAI_XFER_QUEUE_SIZE]`

30.9.2.1.0.70.2 `sai_transfer_t sai_sdma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

30.9.2.1.0.70.3 `volatile uint8_t sai_sdma_handle_t::queueUser`

30.9.3 Function Documentation

30.9.3.1 `void SAI_TransferTxCreateHandleSDMA (I2S_Type * base, sai_sdma_handle_t * handle, sai_sdma_callback_t callback, void * userData, sdma_handle_t * dmaHandle, uint32_t eventSource)`

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 SDMA Driver

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI SDMA 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>	SDMA handle pointer, this handle shall be static allocated by users.

30.9.3.2 void SAI_TransferRxCreateHandleSDMA (I2S_Type * *base*, sai_sdma_handle_t * *handle*, sai_sdma_callback_t *callback*, void * *userData*, sdma_handle_t * *dmaHandle*, uint32_t *eventSource*)

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 SDMA 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>	SDMA handle pointer, this handle shall be static allocated by users.

30.9.3.3 void SAI_TransferTxSetFormatSDMA (I2S_Type * *base*, sai_sdma_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 SDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

<i>handle</i>	SAI SDMA 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.

30.9.3.4 void SAI_TransferRxSetFormatSDMA (I2S_Type * *base*, sai_sdma_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 SDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI SDMA 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.

30.9.3.5 status_t SAI_TransferSendSDMA (I2S_Type * *base*, sai_sdma_handle_t * *handle*, sai_transfer_t * *xfer*)

SAI SDMA 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 SDMA handle pointer.
<i>xfer</i>	Pointer to the DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI SDMA send successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_TxBusy</i>	SAI is busy sending data.

30.9.3.6 **status_t SAI_TransferReceiveSDMA (I2S_Type * *base*, sai_sdma_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 SDMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI SDMA receive successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_RxBusy</i>	SAI is busy receiving data.

30.9.3.7 **void SAI_TransferAbortSendSDMA (I2S_Type * *base*, sai_sdma_handle_t * *handle*)**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI SDMA handle pointer.

30.9.3.8 void SAI_TransferAbortReceiveSDMA (I2S_Type * *base*, sai_sdma_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI SDMA handle pointer.

Chapter 31

SDHC: Secure Digital Host Controller Driver

31.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Secure Digital Host Controller (SDHC) module of MCUXpresso SDK devices.

31.2 Typical use case

31.2.1 SDHC Operation

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

Data Structures

- struct [sdhc_adma2_descriptor_t](#)
Defines the ADMA2 descriptor structure. [More...](#)
- struct [sdhc_capability_t](#)
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.

Typical use case

- #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_PrepAdmaDescriptorFailed = MAKE_STATUS(kStatusGroup_SDHC, 1U),
 kStatus_SDHC_SendCommandFailed = MAKE_STATUS(kStatusGroup_SDHC, 2U),
 kStatus_SDHC_TransferDataFailed = MAKE_STATUS(kStatusGroup_SDHC, 3U),
 kStatus_SDHC_DMADataBufferAddrNotAlign }
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` {

 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` {

Typical use case

```
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 = (int)(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_AdmaLenghMismatchFlag = 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 {
```

```
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 = (int)SDHC_FEVT_CINT_MASK,
```

```
kSDHC_ForceEventDmaError = SDHC_FEVT_DMAE_MASK,
```

```
kSDHC_ForceEventsAll }
```

*Force event bit position.*
- ```
• enum sdhc_data_bus_width_t {
```

```
kSDHC_DataBusWidth1Bit = 0U,
```

```
kSDHC_DataBusWidth4Bit = 1U,
```

```
kSDHC_DataBusWidth8Bit = 2U }
```

Data transfer width.
- ```
• enum sdhc_endian_mode_t {
```

## Typical use case

- ```
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_card_command_type_t {  
    kCARD_CommandTypeNormal = 0U,  
    kCARD_CommandTypeSuspend = 1U,  
    kCARD_CommandTypeResume = 2U,  
    kCARD_CommandTypeAbort = 3U }  
    The command type.  
• enum sdhc_card_response_type_t {  
    kCARD_ResponseNone = 0U,  
    kCARD_ResponseR1 = 1U,  
    kCARD_ResponseR1b = 2U,  
    kCARD_ResponseR2 = 3U,  
    kCARD_ResponseR3 = 4U,  
    kCARD_ResponseR4 = 5U,  
    kCARD_ResponseR5 = 6U,  
    kCARD_ResponseR5b = 7U,  
    kCARD_ResponseR6 = 8U,  
    kCARD_ResponseR7 = 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, 8U))
Driver version 2.1.8.

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.

Typical use case

- 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.
- static void [SDHC_CardDetectByData3](#) (SDHC_Type *base, bool enable)
detect card insert status.
- 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 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_EnableCardDetectTest](#) (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.

31.3 Data Structure Documentation

31.3.1 struct sdhc_adma2_descriptor_t

Data Fields

- uint32_t `attribute`
The control and status field.
- const uint32_t * `address`
The address field.

31.3.2 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)

31.3.3 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 Structure Documentation

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)

31.3.4 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.

31.3.4.0.0.71 Field Documentation

31.3.4.0.0.71.1 uint32_t sdhc_boot_config_t::ackTimeoutCount

The available range is 0 ~ 15.

31.3.4.0.0.71.2 sdhc_boot_mode_t sdhc_boot_config_t::bootMode

31.3.4.0.0.71.3 uint32_t sdhc_boot_config_t::blockCount

Available range is 0 ~ 65535.

31.3.5 struct sdhc_config_t

Data Fields

- bool **cardDetectDat3**

- **sdhc_endian_mode_t endianMode**
Enable DAT3 as card detection pin.
- **sdhc_dma_mode_t dmaMode**
Endian mode.
- **uint32_t readWatermarkLevel**
DMA mode.
- **uint32_t writeWatermarkLevel**
Watermark level for DMA read operation.
- **uint32_t writeWatermarkLevel**
Watermark level for DMA write operation.

31.3.5.0.0.72 Field Documentation

31.3.5.0.0.72.1 uint32_t sdhc_config_t::readWatermarkLevel

Available range is 1 ~ 128.

31.3.5.0.0.72.2 uint32_t sdhc_config_t::writeWatermarkLevel

Available range is 1 ~ 128.

31.3.6 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.

31.3.7 struct sdhc_command_t

Define card command-related attribute.

Data Structure Documentation

Data Fields

- `uint32_t index`
Command index.
- `uint32_t argument`
Command argument.
- `sdhc_card_command_type_t type`
Command type.
- `sdhc_card_response_type_t responseType`
Command response type.
- `uint32_t response [4U]`
Response for this command.
- `uint32_t responseErrorFlags`
response error flag, the flag which need to check the command reponse

31.3.8 struct sdhc_transfer_t

Data Fields

- `sdhc_data_t * data`
Data to transfer.
- `sdhc_command_t * command`
Command to send.

31.3.9 struct sdhc_transfer_callback_t

Data Fields

- `void(* CardInserted)(SDHC_Type *base, void *userData)`
Card inserted occurs when DAT3/CD pin is for card detect.
- `void(* CardRemoved)(SDHC_Type *base, void *userData)`
Card removed occurs.
- `void(* SdioInterrupt)(SDHC_Type *base, void *userData)`
SDIO card interrupt occurs.
- `void(* SdioBlockGap)(SDHC_Type *base, void *userData)`
SDIO card stopped at block gap occurs.
- `void(* TransferComplete)(SDHC_Type *base, sdhc_handle_t *handle, status_t status, void *userData)`
Transfer complete callback.

31.3.10 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.

31.3.11 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.

31.4 Macro Definition Documentation

31.4.1 #define FSL_SDHC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 8U))

Enumeration Type Documentation

31.5 Typedef Documentation

31.5.1 `typedef uint32_t sdhc_adma1_descriptor_t`

31.5.2 `typedef status_t(* sdhc_transfer_function_t)(SDHC_Type *base, sdhc_transfer_t *content)`

31.6 Enumeration Type Documentation

31.6.1 `enum _sdhc_status`

Enumerator

kStatus_SDHC_BusyTransferring Transfer is on-going.

kStatus_SDHC_PrepAdmaDescriptorFailed Set DMA descriptor failed.

kStatus_SDHC_SendCommandFailed Send command failed.

kStatus_SDHC_TransferDataFailed Transfer data failed.

kStatus_SDHC_DMADataBufferAddrNotAlign data buffer addr not align in DMA mode

31.6.2 `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.

31.6.3 `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.

31.6.4 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.

31.6.5 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.

31.6.6 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.

Enumeration Type Documentation

<i>kSDHC_Data1LineLevelFlag</i>	Data1 line signal level.
<i>kSDHC_Data2LineLevelFlag</i>	Data2 line signal level.
<i>kSDHC_Data3LineLevelFlag</i>	Data3 line signal level.
<i>kSDHC_Data4LineLevelFlag</i>	Data4 line signal level.
<i>kSDHC_Data5LineLevelFlag</i>	Data5 line signal level.
<i>kSDHC_Data6LineLevelFlag</i>	Data6 line signal level.
<i>kSDHC_Data7LineLevelFlag</i>	Data7 line signal level.

31.6.7 enum _sdhc_interrupt_status_flag

Enumerator

<i>kSDHC_CommandCompleteFlag</i>	Command complete.
<i>kSDHC_DataCompleteFlag</i>	Data complete.
<i>kSDHC_BlockGapEventFlag</i>	Block gap event.
<i>kSDHC_DmaCompleteFlag</i>	DMA interrupt.
<i>kSDHC_BufferWriteReadyFlag</i>	Buffer write ready.
<i>kSDHC_BufferReadReadyFlag</i>	Buffer read ready.
<i>kSDHC_CardInsertionFlag</i>	Card inserted.
<i>kSDHC_CardRemovalFlag</i>	Card removed.
<i>kSDHC_CardInterruptFlag</i>	Card interrupt.
<i>kSDHC_CommandTimeoutFlag</i>	Command timeout error.
<i>kSDHC_CommandCrcErrorFlag</i>	Command CRC error.
<i>kSDHC_CommandEndBitErrorFlag</i>	Command end bit error.
<i>kSDHC_CommandIndexErrorFlag</i>	Command index error.
<i>kSDHC_DataTimeoutFlag</i>	Data timeout error.
<i>kSDHC_DataCrcErrorFlag</i>	Data CRC error.
<i>kSDHC_DataEndBitErrorFlag</i>	Data end bit error.
<i>kSDHC_AutoCommand12ErrorFlag</i>	Auto CMD12 error.
<i>kSDHC_DmaErrorFlag</i>	DMA error.
<i>kSDHC_CommandErrorFlag</i>	Command error.
<i>kSDHC_DataErrorFlag</i>	Data error.
<i>kSDHC_ErrorFlag</i>	All error.
<i>kSDHC_DataFlag</i>	Data interrupts.
<i>kSDHC_CommandFlag</i>	Command interrupts.
<i>kSDHC_CardDetectFlag</i>	Card detection interrupts.
<i>kSDHC_AllInterruptFlags</i>	All flags mask.

31.6.8 enum _sdhc_auto_command12_error_status_flag

Enumerator

<i>kSDHC_AutoCommand12NotExecutedFlag</i>	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.

31.6.9 enum _sdhc_adma_error_status_flag

Enumerator

kSDHC_AdmaLengthMismatchFlag Length mismatch error.
kSDHC_AdmaDescriptorErrorFlag Descriptor error.

31.6.10 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.

31.6.11 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.

Enumeration Type Documentation

kSDHC_ForceEventCardInt Card interrupt.

kSDHC_ForceEventDmaError Dma error.

kSDHC_ForceEventsAll All force event flags mask.

31.6.12 enum sdhc_data_bus_width_t

Enumerator

kSDHC_DataBusWidth1Bit 1-bit mode

kSDHC_DataBusWidth4Bit 4-bit mode

kSDHC_DataBusWidth8Bit 8-bit mode

31.6.13 enum sdhc_endian_mode_t

Enumerator

kSDHC_EndianModeBig Big endian mode.

kSDHC_EndianModeHalfWordBig Half word big endian mode.

kSDHC_EndianModeLittle Little endian mode.

31.6.14 enum sdhc_dma_mode_t

Enumerator

kSDHC_DmaModeNo No DMA.

kSDHC_DmaModeAdma1 ADMA1 is selected.

kSDHC_DmaModeAdma2 ADMA2 is selected.

31.6.15 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.

31.6.16 enum sdhc_boot_mode_t

Enumerator

kSDHC_BootModeNormal Normal boot.
kSDHC_BootModeAlternative Alternative boot.

31.6.17 enum sdhc_card_command_type_t

Enumerator

kCARD_CommandTypeNormal Normal command.
kCARD_CommandTypeSuspend Suspend command.
kCARD_CommandTypeResume Resume command.
kCARD_CommandTypeAbort Abort command.

31.6.18 enum sdhc_card_response_type_t

Define the command response type from card to host controller.

Enumerator

kCARD_ResponseNone Response type: none.
kCARD_ResponseR1 Response type: R1.
kCARD_ResponseR1b Response type: R1b.
kCARD_ResponseR2 Response type: R2.
kCARD_ResponseR3 Response type: R3.
kCARD_ResponseR4 Response type: R4.
kCARD_ResponseR5 Response type: R5.
kCARD_ResponseR5b Response type: R5b.
kCARD_ResponseR6 Response type: R6.
kCARD_ResponseR7 Response type: R7.

31.6.19 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.

Function Documentation

kSDHC_Adma1DescriptorTypeNop No operation.
kSDHC_Adma1DescriptorTypeTransfer Transfer data.
kSDHC_Adma1DescriptorTypeLink Link descriptor.
kSDHC_Adma1DescriptorTypeSetLength Set data length.

31.6.20 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.

31.7 Function Documentation

31.7.1 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.
------------------------	-----------------------

31.7.2 void SDHC_Deinit (**SDHC_Type** * *base*)

Parameters

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

31.7.3 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.

31.7.4 **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*)

Parameters

<i>base</i>	SDHC peripheral base address.
<i>dmaMode</i>	DMA mode.
<i>table</i>	ADMA table address.

Function Documentation

<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.

31.7.5 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).

31.7.6 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).

31.7.7 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).
-------------	---

31.7.8 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).

31.7.9 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).

31.7.10 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).

31.7.11 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).

31.7.12 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).

31.7.13 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).

31.7.14 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.

**31.7.15 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.

**31.7.16 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.

31.7.17 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.

31.7.18 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.

31.7.19 static void SDHC_CardDetectByData3 (SDHC_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>enable/disable</i>	flag

31.7.20 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.

31.7.21 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.

31.7.22 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.

31.7.23 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

Function Documentation

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

Returns

The data has been read.

31.7.24 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.

31.7.25 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.

31.7.26 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.
-------------	--

31.7.27 void SDHC_EnableSdioControl (SDHC_Type * *base*, uint32_t *mask*, bool *enable*)

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	SDIO card control flags mask(_sdhc_sdio_control_flag).
<i>enable</i>	True to enable, false to disable.

31.7.28 static void SDHC_SetContinueRequest (SDHC_Type * *base*) [inline], [static]

Parameters

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

31.7.29 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

Function Documentation

<i>base</i>	SDHC peripheral base address.
<i>config</i>	The MMC boot configuration information.

**31.7.30 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).

31.7.31 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. This function support non word align data addr transfer support, if data buffer addr is not align in DMA mode, the API will continue finish the transfer by polling IO directly. 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_PrepareAdmaDescriptorFailed</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.

31.7.32 void SDHC_TransferCreateHandle (**SDHC_Type** * *base*, **sdhc_handle_t** * *handle*, const **sdhc_transfer_callback_t** * *callback*, **void** * *userData*)

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.

31.7.33 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. This function support non word align data addr transfer support, if data buffer addr is not align in DMA mode, the API will continue finish the transfer by polling IO directly. 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.

Function Documentation

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_Prepares-AdmaDescriptorFailed</i>	Prepare ADMA descriptor failed.
<i>kStatus_Success</i>	Operate successfully.

31.7.34 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.

Chapter 32

SIM: System Integration Module Driver

32.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Integration Module (SIM) of MCUXpresso SDK devices.

Data Structures

- struct `sim_uid_t`
Unique ID. [More...](#)

Enumerations

- enum `_sim_usb_volt_reg_enable_mode` {
 `kSIM_UsbVoltRegEnable` = (int)`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, 1, 0)`)
Driver version 2.0.0.

Function Documentation

32.2 Data Structure Documentation

32.2.1 struct sim_uid_t

Data Fields

- uint32_t **MH**
UIDMH.
- uint32_t **ML**
UIDML.
- uint32_t **L**
UIDL.

32.2.1.0.0.73 Field Documentation

32.2.1.0.0.73.1 uint32_t sim_uid_t::MH

32.2.1.0.0.73.2 uint32_t sim_uid_t::ML

32.2.1.0.0.73.3 uint32_t sim_uid_t::L

32.3 Enumeration Type Documentation

32.3.1 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.

32.3.2 enum _sim_flash_mode

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.

kSIM_FlashDisable Disable flash in normal mode.

32.4 Function Documentation

32.4.1 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);
```

Function Documentation

Parameters

<i>mask</i>	USB voltage regulator enable setting.
-------------	---------------------------------------

32.4.2 void SIM_GetUniqueId (sim_uid_t * *uid*)

Parameters

<i>uid</i>	Pointer to the structure to save the UID value.
------------	---

32.4.3 static void SIM_SetFlashMode (uint8_t *mode*) [inline], [static]

Parameters

<i>mode</i>	The mode to set; see <u>_sim_flash_mode</u> for mode details.
-------------	---

Chapter 33

SMC: System Mode Controller Driver

33.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Mode Controller (SMC) module of MCUXpresso SDK devices. The SMC module sequences the system in and out of all low-power stop and run modes.

API functions are provided to configure the system for working in a dedicated power mode. For different power modes, SMC_SetPowerModeXXX() function accepts different parameters. System power mode state transitions are not available between power modes. For details about available transitions, see the power mode transitions section in the SoC reference manual.

33.2 Typical use case

33.2.1 Enter wait or stop modes

SMC driver provides APIs to set MCU to different wait modes and stop modes. Pre and post functions are used for setting the modes. The pre functions and post functions are used as follows.

Disable/enable the interrupt through PRIMASK. This is an example use case. The application sets the wakeup interrupt and calls SMC function [SMC_SetPowerModeStop](#) to set the MCU to STOP mode, but the wakeup interrupt happens so quickly that the ISR completes before the function [SMC_SetPowerModeStop](#). As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMASK. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.

```
SMC_PreEnterStopModes();  
/* Enable the wakeup interrupt here. */  
SMC_SetPowerModeStop(SMC, kSMC_PartialStop);  
SMC_PostExitStopModes();
```

For legacy Kinetis, when entering stop modes, the flash speculation might be interrupted. As a result, the prefetched code or data might be broken. To make sure the flash is idle when entering the stop modes, smc driver allocates a RAM region, the code to enter stop modes are executed in RAM, thus the flash is idle and no prefetch is performed while entering stop modes. Application should make sure that, the rw data of fsl_smci.c is located in memory region which is not powered off in stop modes, especially LLS2 modes.

For STOP, VLPS, and LLS3, the whole RAM are powered up, so after woken up, the RAM function could continue executing. For VLLS mode, the system resets after woken up, the RAM content might be re-initialized. For LLS2 mode, only part of RAM are powered on, so application must make sure that, the

Typical use case

rw data of fsl_smcc.c is located in memory region which is not powered off, otherwise after woken up, the MCU could not get right code to execute.

Data Structures

- struct `smc_power_mode_vlls_config_t`
SMC Very Low-Leakage Stop power mode configuration. [More...](#)

Enumerations

- enum `smc_power_mode_protection_t`{
 kSMC_AllowPowerModeVlls = SMC_PMPROT_AVLLS_MASK,
 kSMC_AllowPowerModeLls = SMC_PMPROT_ALLS_MASK,
 kSMC_AllowPowerModeVlp = SMC_PMPROT_AVLP_MASK,
 kSMC_AllowPowerModeAll }
Power Modes Protection.
- enum `smc_power_state_t`{
 kSMC_PowerStateRun = 0x01U << 0U,
 kSMC_PowerStateStop = 0x01U << 1U,
 kSMC_PowerStateVlpr = 0x01U << 2U,
 kSMC_PowerStateVlpw = 0x01U << 3U,
 kSMC_PowerStateVlps = 0x01U << 4U,
 kSMC_PowerStateLls = 0x01U << 5U,
 kSMC_PowerStateVlls = 0x01U << 6U }
Power Modes in PMSTAT.
- enum `smc_run_mode_t`{
 kSMC_RunNormal = 0U,
 kSMC_RunVlpr = 2U }
Run mode definition.
- enum `smc_stop_mode_t`{
 kSMC_StopNormal = 0U,
 kSMC_StopVlps = 2U,
 kSMC_StopLls = 3U,
 kSMC_StopVlls = 4U }
Stop mode definition.
- enum `smc_stop_submode_t`{
 kSMC_StopSub0 = 0U,
 kSMC_StopSub1 = 1U,
 kSMC_StopSub2 = 2U,
 kSMC_StopSub3 = 3U }
VLLS/LLS stop sub mode definition.
- enum `smc_partial_stop_option_t`{
 kSMC_PartialStop = 0U,
 kSMC_PartialStop1 = 1U,
 kSMC_PartialStop2 = 2U }
Partial STOP option.
- enum `_smc_status` { kStatus_SMC_StopAbort = MAKE_STATUS(kStatusGroup_POWER, 0) }

SMC configuration status.

Driver version

- #define **FSL_SMC_DRIVER_VERSION** (MAKE_VERSION(2, 0, 4))
SMC driver version 2.0.4.

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)
Recover after wake up from stop modes.
- void **SMC_PreEnterWaitModes** (void)
Prepares to enter wait modes.
- void **SMC_PostExitWaitModes** (void)
Recover after wake up from stop modes.
- status_t **SMC_SetPowerModeRun** (SMC_Type *base)
Configures the system to RUN power mode.
- status_t **SMC_SetPowerModeWait** (SMC_Type *base)
Configures the system to WAIT power mode.
- status_t **SMC_SetPowerModeStop** (SMC_Type *base, **smc_partial_stop_option_t** option)
Configures the system to Stop power mode.
- status_t **SMC_SetPowerModeVlpr** (SMC_Type *base, bool wakeupMode)
Configures the system to VLPR power mode.
- status_t **SMC_SetPowerModeVlpw** (SMC_Type *base)
Configures the system to VLPW power mode.
- status_t **SMC_SetPowerModeVlps** (SMC_Type *base)
Configures the system to VLPS power mode.
- status_t **SMC_SetPowerModeLls** (SMC_Type *base)
Configures the system to LLS power mode.
- status_t **SMC_SetPowerModeVlls** (SMC_Type *base, const **smc_power_mode_vlls_config_t** *config)
Configures the system to VLLS power mode.

33.3 Data Structure Documentation

33.3.1 struct smc_power_mode_vlls_config_t

Data Fields

- **smc_stop_submode_t subMode**
Very Low-leakage Stop sub-mode.
- **bool enablePorDetectInVlls0**
Enable Power on reset detect in VLLS mode.

Enumeration Type Documentation

33.4 Macro Definition Documentation

33.4.1 `#define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))`

33.5 Enumeration Type Documentation

33.5.1 enum smc_power_mode_protection_t

Enumerator

kSMC_AllowPowerModeVlls Allow Very-low-leakage Stop Mode.

kSMC_AllowPowerModeLls Allow Low-leakage Stop Mode.

kSMC_AllowPowerModeVlp Allow Very-Low-power Mode.

kSMC_AllowPowerModeAll Allow all power mode.

33.5.2 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

33.5.3 enum smc_run_mode_t

Enumerator

kSMC_RunNormal Normal RUN mode.

kSMC_RunVlpr Very-low-power RUN mode.

33.5.4 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.

33.5.5 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.

33.5.6 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.

33.5.7 enum _smc_status

Enumerator

- kStatus_SMC_StopAbort* Entering Stop mode is abort.

33.6 Function Documentation

33.6.1 static void SMC_SetPowerModeProtection (SMC_Type * *base*, uint8_t *allowedModes*) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the smc_power_mode_protection_t. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map. For example, to allow LLS and VLLS, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps). To allow all modes, use SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll).

Parameters

Function Documentation

<i>base</i>	SMC peripheral base address.
<i>allowedModes</i>	Bitmap of the allowed power modes.

33.6.2 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.

33.6.3 void SMC_PreEnterStopModes (void)

This function should be called before entering STOP/VLPS/LLS/VLLS modes.

33.6.4 void SMC_PostExitStopModes (void)

This function should be called after wake up from STOP/VLPS/LLS/VLLS modes. It is used with [SMC_PreEnterStopModes](#).

33.6.5 void SMC_PreEnterWaitModes (void)

This function should be called before entering WAIT/VLPW modes.

33.6.6 void SMC_PostExitWaitModes (void)

This function should be called after wake up from WAIT/VLPW modes. It is used with [SMC_PreEnterWaitModes](#).

33.6.7 **status_t SMC_SetPowerModeRun (SMC_Type * *base*)**

Function Documentation

Parameters

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

Returns

SMC configuration error code.

33.6.8 **status_t SMC_SetPowerModeWait (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

33.6.9 **status_t SMC_SetPowerModeStop (SMC_Type * *base*, smc_partial_stop_option_t *option*)**

Parameters

<i>base</i>	SMC peripheral base address.
<i>option</i>	Partial Stop mode option.

Returns

SMC configuration error code.

33.6.10 **status_t SMC_SetPowerModeVlpr (SMC_Type * *base*, bool *wakeupMode*)**

Parameters

<i>base</i>	SMC peripheral base address.
<i>wakeupMode</i>	Enter Normal Run mode if true, else stay in VLPR mode.

Returns

SMC configuration error code.

33.6.11 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

33.6.12 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

33.6.13 status_t SMC_SetPowerModeLls (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

33.6.14 status_t SMC_SetPowerModeVlls (SMC_Type * *base*, const smc_power_mode_vlls_config_t * *config*)

Function Documentation

Parameters

<i>base</i>	SMC peripheral base address.
<i>config</i>	The VLLS power mode configuration structure.

Returns

SMC configuration error code.

Chapter 34

UART: Universal Asynchronous Receiver/Transmitter Driver

34.1 Overview

Modules

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

34.2 UART Driver

34.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of MCUXpresso SDK devices.

The UART driver includes functional APIs and transactional APIs.

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

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `uart_handle_t` as the second parameter. Initialize the handle by calling the [UART_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer, which means that the functions [UART_TransferSendNonBlocking\(\)](#) and [UART_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_UART_TxIdle` and `kStatus_UART_RxIdle`.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the [UART_TransferCreateHandle\(\)](#). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The [UART_TransferReceiveNonBlocking\(\)](#) function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the `kStatus_UART_RxIdle`.

If the receive ring buffer is full, the upper layer is informed through a callback with the `kStatus_UART_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.

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/uart. In this example, the buffer size is 32, but only 31 bytes are used for saving data.

34.2.2 Typical use case

34.2.2.1 UART Send/receive using a polling method

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

34.2.2.2 UART Send/receive using an interrupt method

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

34.2.2.3 UART Receive using the ringbuffer feature

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

34.2.2.4 UART Send/Receive using the DMA method

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

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,
 kStatus_UART_IdleLineDetected = MAKE_STATUS(kStatusGroup_UART, 14) }

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_idle_type_select_t` {
 kUART_IdleTypeStartBit = 0U,
 kUART_IdleTypeStopBit = 1U }
- UART idle type select.*
- 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` {

`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` = (int)(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.*

Functions

- `uint32_t UARTGetInstance (UART_Type *base)`
Get the UART instance from peripheral base address.

Driver version

- `#define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 6))`
UART driver version 2.1.6.

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.

UART Driver

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.
- size_t **UART_TransferGetRxRingBufferLength** (uart_handle_t *handle)

Get the length of received data in RX ring buffer.
- status_t **UART_TransferSendNonBlocking** (UART_Type *base, uart_handle_t *handle, **uart_transfer_t** *xfer)

Transmits a buffer of data using the interrupt method.
- void **UART_TransferAbortSend** (UART_Type *base, uart_handle_t *handle)

Aborts the interrupt-driven data transmit.
- status_t **UART_TransferGetSendCount** (UART_Type *base, uart_handle_t *handle, uint32_t *count)

Gets the number of bytes written to the UART TX register.
- status_t **UART_TransferReceiveNonBlocking** (UART_Type *base, uart_handle_t *handle, **uart_transfer_t** *xfer, size_t *receivedBytes)

Receives a buffer of data using an interrupt method.
- void **UART_TransferAbortReceive** (UART_Type *base, uart_handle_t *handle)

Aborts the interrupt-driven data receiving.
- status_t **UART_TransferGetReceiveCount** (UART_Type *base, uart_handle_t *handle, uint32_t *count)

Gets the number of bytes that have been received.
- void **UART_TransferHandleIRQ** (UART_Type *base, uart_handle_t *handle)

UART IRQ handle function.
- void **UART_TransferHandleErrorIRQ** (UART_Type *base, uart_handle_t *handle)

UART Error IRQ handle function.

34.2.3 Data Structure Documentation

34.2.3.1 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 **enableRxRTS**

RX RTS enable.
- bool **enableTxCTS**

TX CTS enable.
- **uart_idle_type_select_t** **idleType**

UART Driver

IDLE type select.

- bool **enableTx**
Enable TX.
- bool **enableRx**
Enable RX.

34.2.3.1.0.74 Field Documentation

34.2.3.1.0.74.1 **uart_idle_type_select_t uart_config_t::idleType**

34.2.3.2 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.

34.2.3.2.0.75 Field Documentation

34.2.3.2.0.75.1 **uint8_t* uart_transfer_t::data**

34.2.3.2.0.75.2 **size_t uart_transfer_t::dataSize**

34.2.3.3 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

34.2.3.3.0.76 Field Documentation

- 34.2.3.3.0.76.1 `uint8_t* volatile uart_handle_t::txData`
- 34.2.3.3.0.76.2 `volatile size_t uart_handle_t::txDataSize`
- 34.2.3.3.0.76.3 `size_t uart_handle_t::txDataSizeAll`
- 34.2.3.3.0.76.4 `uint8_t* volatile uart_handle_t::rxData`
- 34.2.3.3.0.76.5 `volatile size_t uart_handle_t::rxDataSize`
- 34.2.3.3.0.76.6 `size_t uart_handle_t::rxDataSizeAll`
- 34.2.3.3.0.76.7 `uint8_t* uart_handle_t::rxRingBuffer`
- 34.2.3.3.0.76.8 `size_t uart_handle_t::rxRingBufferSize`
- 34.2.3.3.0.76.9 `volatile uint16_t uart_handle_t::rxRingBufferHead`
- 34.2.3.3.0.76.10 `volatile uint16_t uart_handle_t::rxRingBufferTail`
- 34.2.3.3.0.76.11 `uart_transfer_callback_t uart_handle_t::callback`
- 34.2.3.3.0.76.12 `void* uart_handle_t::userData`
- 34.2.3.3.0.76.13 `volatile uint8_t uart_handle_t::txState`

34.2.4 Macro Definition Documentation

- 34.2.4.1 `#define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 6))`

34.2.5 Typedef Documentation

- 34.2.5.1 `typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)`

34.2.6 Enumeration Type Documentation

34.2.6.1 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.
kStatus_UART_IdleLineDetected UART IDLE line detected.

34.2.6.2 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.

34.2.6.3 enum uart_stop_bit_count_t

Enumerator

kUART_OneStopBit One stop bit.
kUART_TwoStopBit Two stop bits.

34.2.6.4 enum uart_idle_type_select_t

Enumerator

kUART_IdleTypeStartBit Start counting after a valid start bit.
kUART_IdleTypeStopBit Start counting after a stop bit.

34.2.6.5 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.

34.2.6.6 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.

34.2.7 Function Documentation

34.2.7.1 uint32_t **UART_GetInstance** (**UART_Type * base**)

Parameters

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

Returns

UART instance.

34.2.7.2 **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

34.2.7.3 **void UART_Deinit(UART_Type * *base*)**

This function waits for TX complete, disables TX and RX, and disables the UART clock.

UART Driver

Parameters

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

34.2.7.4 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->idleType* = kUART_IdleTypeStartBit; *uartConfig->enableTx* = false; *uartConfig->enableRx* = false;

Parameters

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

34.2.7.5 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);  
*
```

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.
------------------------	-------------------------

34.2.7.6 `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`.

34.2.7.7 `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 <code>_uart_flags</code> .

Return values

UART Driver

<i>kStatus_UART_Flag_CannotClearManually</i>	The flag can't be cleared by this function but it is cleared automatically by hardware.
<i>kStatus_Success</i>	Status in the mask is cleared.

34.2.7.8 void **UART_EnableInterrupts** (**UART_Type * base**, **uint32_t mask**)

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

```
*     UART_EnableInterrupts(UART1,  
    kUART_TxDataRegEmptyInterruptEnable |  
    kUART_RxDataRegFullInterruptEnable);  
*
```

Parameters

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

34.2.7.9 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);  
*
```

Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of _uart_interrupt_enable .

34.2.7.10 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](#).

34.2.7.11 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.

34.2.7.12 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.

34.2.7.13 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.

UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>enable</i>	True to enable, false to disable.

34.2.7.14 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.

34.2.7.15 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.

34.2.7.16 static void UART_WriteByte (**UART_Type** * *base*, **uint8_t** *data*) [inline], [static]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

<i>base</i>	UART peripheral base address.
<i>data</i>	The byte to write.

34.2.7.17 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.

34.2.7.18 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.

34.2.7.19 **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.

UART Driver

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.

34.2.7.20 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.

34.2.7.21 void **UART_TransferStartRingBuffer** (**UART_Type * base**, **uart_handle_t * handle**, **uint8_t * ringBuffer**, **size_t ringBufferSize**)

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the [UART_TransferReceiveNonBlocking\(\)](#) API. If data is already received in the ring buffer, the user can get the received data from the ring buffer directly.

Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>ringBuffer</i>	Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
<i>ringBufferSize</i>	Size of the ring buffer.

34.2.7.22 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.

34.2.7.23 size_t **UART_TransferGetRxRingBufferLength** (**uart_handle_t * handle**)

Parameters

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

Returns

Length of received data in RX ring buffer.

34.2.7.24 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.

UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART transfer structure. See uart_transfer_t .

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.

34.2.7.25 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.

34.2.7.26 **status_t** **UART_TransferGetSendCount** (**UART_Type** * *base*, **uart_handle_t** * *handle*, **uint32_t** * *count*)

This function gets the number of bytes written to the UART TX register by using the interrupt method.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Send bytes count.

Return values

<i>kStatus_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	The parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

34.2.7.27 `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 uart_transfer_t .
<i>receivedBytes</i>	Bytes received from the ring buffer directly.

Return values

<i>kStatus_Success</i>	Successfully queue the transfer into transmit queue.
<i>kStatus_UART_RxBusy</i>	Previous receive request is not finished.
<i>kStatus_InvalidArgument</i>	Invalid argument.

34.2.7.28 `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.

UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

34.2.7.29 **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> ;

34.2.7.30 **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.

34.2.7.31 **void UART_TransferHandleErrorIRQ (*UART_Type* * *base*, *uart_handle_t* * *handle*)**

This function handles the UART error IRQ request.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

34.3 UART DMA Driver

34.3.1 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.

Driver version

- #define [FSL_UART_DMA_DRIVER_VERSION](#) (MAKE_VERSION(2, 1, 6))
UART DMA driver version 2.1.6.

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.

34.3.2 Data Structure Documentation

34.3.2.1 struct _uart_dma_handle

Data Fields

- `UART_Type *base`
UART peripheral base address.
- `uart_dma_transfer_callback_t callback`
Callback function.
- `void *userData`
UART callback function parameter.
- `size_t rxDataSizeAll`
Size of the data to receive.
- `size_t txDataSizeAll`
Size of the data to send out.
- `dma_handle_t *txDmaHandle`
The DMA TX channel used.
- `dma_handle_t *rxDmaHandle`
The DMA RX channel used.
- `volatile uint8_t txState`
TX transfer state.
- `volatile uint8_t rxState`
RX transfer state.

UART DMA Driver

34.3.2.1.0.77 Field Documentation

34.3.2.1.0.77.1 `UART_Type* uart_dma_handle_t::base`

34.3.2.1.0.77.2 `uart_dma_transfer_callback_t uart_dma_handle_t::callback`

34.3.2.1.0.77.3 `void* uart_dma_handle_t::userData`

34.3.2.1.0.77.4 `size_t uart_dma_handle_t::rxDataSizeAll`

34.3.2.1.0.77.5 `size_t uart_dma_handle_t::txDataSizeAll`

34.3.2.1.0.77.6 `dma_handle_t* uart_dma_handle_t::txDmaHandle`

34.3.2.1.0.77.7 `dma_handle_t* uart_dma_handle_t::rxDmaHandle`

34.3.2.1.0.77.8 `volatile uint8_t uart_dma_handle_t::txState`

34.3.3 Macro Definition Documentation

34.3.3.1 `#define FSL_UART_DMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 6))`

34.3.4 Typedef Documentation

34.3.4.1 `typedef void(* uart_dma_transfer_callback_t)(UART_Type *base,
uart_dma_handle_t *handle, status_t status, void *userData)`

34.3.5 Function Documentation

34.3.5.1 `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.

34.3.5.2 `status_t UART_TransferSendDMA (UART_Type * base, uart_dma_handle_t * handle, uart_transfer_t * xfer)`

This function sends data using DMA. This is non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART DMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

34.3.5.3 `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 uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

34.3.5.4 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.

34.3.5.5 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.

34.3.5.6 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> ;

34.3.5.7 **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> ;

34.4 UART eDMA Driver

34.4.1 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.

Driver version

- [#define FSL_UART_EDMA_DRIVER_VERSION \(MAKE_VERSION\(2, 1, 6\)\)](#)
UART EDMA driver version 2.1.6.

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.

34.4.2 Data Structure Documentation

34.4.2.1 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.

UART eDMA Driver

34.4.2.1.0.78 Field Documentation

34.4.2.1.0.78.1 `uart_edma_transfer_callback_t uart_edma_handle_t::callback`

34.4.2.1.0.78.2 `void* uart_edma_handle_t::userData`

34.4.2.1.0.78.3 `size_t uart_edma_handle_t::rxDataSizeAll`

34.4.2.1.0.78.4 `size_t uart_edma_handle_t::txDataSizeAll`

34.4.2.1.0.78.5 `edma_handle_t* uart_edma_handle_t::txEdmaHandle`

34.4.2.1.0.78.6 `edma_handle_t* uart_edma_handle_t::rxEdmaHandle`

34.4.2.1.0.78.7 `uint8_t uart_edma_handle_t::nbytes`

34.4.2.1.0.78.8 `volatile uint8_t uart_edma_handle_t::txState`

34.4.3 Macro Definition Documentation

34.4.3.1 `#define FSL_UART_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 6))`

34.4.4 Typedef Documentation

34.4.4.1 `typedef void(* uart_edma_transfer_callback_t)(UART_Type *base,
uart_edma_handle_t *handle, status_t status, void *userData)`

34.4.5 Function Documentation

34.4.5.1 `void UART_TransferCreateHandleEDMA (UART_Type * base,
uart_edma_handle_t * handle, uart_edma_transfer_callback_t callback, void *
userData, edma_handle_t * txEdmaHandle, edma_handle_t * rxEdmaHandle)`

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.
<i>callback</i>	UART callback, NULL means no callback.
<i>userData</i>	User callback function data.
<i>rxEdmaHandle</i>	User-requested DMA handle for RX DMA transfer.
<i>txEdmaHandle</i>	User-requested DMA handle for TX DMA transfer.

34.4.5.2 `status_t UART_SendEDMA (UART_Type * base, uart_edma_handle_t * handle, uart_transfer_t * xfer)`

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART eDMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

34.4.5.3 `status_t UART_ReceiveEDMA (UART_Type * base, uart_edma_handle_t * handle, uart_transfer_t * xfer)`

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

UART eDMA Driver

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.
<i>xfer</i>	UART eDMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

34.4.5.4 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.

34.4.5.5 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.

34.4.5.6 `status_t UART_TransferGetSendCountEDMA` (`UART_Type * base, uart_edma_handle_t * handle, uint32_t * count`)

This function gets the number of bytes that have been written to UART TX register by DMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Send bytes count.

Return values

<i>kStatus_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> ;

34.4.5.7 **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> ;

UART FreeRTOS Driver

34.5 UART FreeRTOS Driver

34.5.1 Overview

Data Structures

- struct `uart_rtos_config_t`
UART configuration structure. [More...](#)

Driver version

- #define `FSL_UART_FREERTOS_DRIVER_VERSION` (MAKE_VERSION(2, 1, 6))
UART freertos driver version 2.1.6.

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.

34.5.2 Data Structure Documentation

34.5.2.1 struct `uart_rtos_config_t`

Data Fields

- `UART_Type` * `base`
UART base address.
- `uint32_t` `srefclk`
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.

34.5.3 Macro Definition Documentation

34.5.3.1 `#define FSL_UART_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 1, 6))`

34.5.4 Function Documentation

34.5.4.1 `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.

34.5.4.2 `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.
---------------------	-----------------------

34.5.4.3 `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.

UART FreeRTOS Driver

Parameters

<i>handle</i>	The RTOS UART handle.
<i>buffer</i>	The pointer to the buffer to send.
<i>length</i>	The number of bytes to send.

34.5.4.4 `int UART_RTOS_Receive (uart_rtos_handle_t * handle, uint8_t * buffer, uint32_t length, size_t * received)`

This function receives data from UART. It is a synchronous API. If data is immediately available, it is returned immediately and the number of bytes received.

Parameters

<i>handle</i>	The RTOS UART handle.
<i>buffer</i>	The pointer to the buffer to write received data.
<i>length</i>	The number of bytes to receive.
<i>received</i>	The pointer to a variable of size_t where the number of received data is filled.

Chapter 35

VREF: Voltage Reference Driver

35.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Crossbar Voltage Reference (VREF) block of MCUXpresso SDK 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.

35.2 Typical use case and example

This example shows how to generate a reference voltage by using the VREF module.

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

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.

Driver version

- #define `FSL_VREF_DRIVER_VERSION` (MAKE_VERSION(2, 1, 0))
Version 2.1.0.

Function Documentation

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.

35.3 Data Structure Documentation

35.3.1 struct vref_config_t

Data Fields

- vref_buffer_mode_t bufferMode
Buffer mode selection.

35.4 Macro Definition Documentation

35.4.1 #define FSL_VREF_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

35.5 Enumeration Type Documentation

35.5.1 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.

35.6 Function Documentation

35.6.1 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.

35.6.2 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.
-------------	--------------------------

35.6.3 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.
---------------	--

35.6.4 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)).

35.6.5 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 36

WDOG: Watchdog Timer Driver

36.1 Overview

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

36.2 Typical use case

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

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.
- enum `wdog_tested_byte_t`{
 `kWDOG_TestByte0` = 0U,
 `kWDOG_TestByte1` = 1U,
 `kWDOG_TestByte2` = 2U,

Typical use case

`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)`
Disables the WDOG module.
- `static void WDOG_EnableInterrupts (WDOG_Type *base, uint32_t mask)`
Enables the WDOG interrupt.
- `static void WDOG_DisableInterrupts (WDOG_Type *base, uint32_t mask)`
Disables 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.

36.3 Data Structure Documentation

36.3.1 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.

36.3.2 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`
Update write-once register enable.
- `bool enableInterrupt`
Enables or disables WDOG interrupt.
- `bool enableWindowMode`
Enables or disables WDOG window mode.

Enumeration Type Documentation

- `uint32_t windowValue`
Window value.
- `uint32_t timeoutValue`
Timeout value.

36.3.3 `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.

36.4 Macro Definition Documentation

36.4.1 `#define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

36.5 Enumeration Type Documentation

36.5.1 `enum wdog_clock_source_t`

Enumerator

kWDOG_LpoClockSource WDOG clock sourced from LPO.

kWDOG_AlternateClockSource WDOG clock sourced from alternate clock source.

36.5.2 `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.

36.5.3 enum wdog_test_mode_t

Enumerator

kWDOG_QuickTest Selects quick test.

kWDOG_Bytetest Selects byte test.

36.5.4 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.

36.5.5 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.

36.5.6 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.

36.6 Function Documentation

36.6.1 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](#)

36.6.2 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

36.6.3 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.

36.6.4 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

36.6.5 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
-------------	------------------------------

36.6.6 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
-------------	------------------------------

36.6.7 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">• kWDOG_InterruptEnable

36.6.8 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">• kWDOG_InterruptEnable

36.6.9 **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.

36.6.10 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

36.6.11 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.

36.6.12 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.

36.6.13 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
-------------	------------------------------

36.6.14 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
-------------	------------------------------

**36.6.15 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.

36.6.16 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 37

Clock Driver

37.1 Overview

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

Modules

- Multipurpose Clock Generator (MCG)

Multipurpose Clock Generator (MCG)

37.2 Multipurpose Clock Generator (MCG)

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

37.2.1 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.

37.2.1.1 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.

37.2.1.2 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 microseconds wait. The function CLOCK_SetExternalRefClkConfig() implements a for loop delay internally. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 50 micro seconds delay. However, when the system clock is slow, the delay time may significantly increase. This for loop count can be optimized for better performance for specific cases.

The MCGPLLCLK is disabled in FBE/FEE/FBI/FEI modes by default. Applications can enable the MCGPLLCLK in these modes using the functions CLOCK_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.

37.2.1.3 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.

37.2.1.4 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.

37.2.1.5 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.

37.2.1.6 MCG mode functions

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

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

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

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

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

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

Multipurpose Clock Generator (MCG)

37.2.2 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.

37.2.2.1 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>

37.2.2.2 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>

37.2.2.3 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)

37.2.2.4 Switch between BLPE and PEE

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

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

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

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

Multipurpose Clock Generator (MCG)

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

37.2.2.5 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(...)

	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	FBI -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	FEE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

37.2.2.6 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)

37.2.3 Code Configuration Option

37.2.3.1 MCG_USER_CONFIG_FLL_STABLE_DELAY_EN

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

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/mcg. Once user is willing to create his own delay function, just assert the macro MCG_USER_CONFIG_FLL_STABLE_DELAY_EN, and then define function CLOCK_FllStableDelay in the application code.

Chapter 38

DMA Manager

38.1 Overview

DMA Manager provides a series of functions to manage the DMAMUX instances and channels.

38.2 Function groups

38.2.1 DMAMGR Initialization and De-initialization

This function group initializes and deinitializes the DMA Manager.

38.2.2 DMAMGR Operation

This function group requests/releases the DMAMUX channel and configures the channel request source.

38.3 Typical use case

38.3.1 DMAMGR static channel allocattion

```
uint8_t channel;
dmamanager_handle_t dmamanager_handle;

/* Initialize DMAMGR */
DMAMGR_Init(&dmamanager_handle, EXAMPLE_DMA_BASEADDR, DMA_CHANNEL_NUMBER, startChannel);
/* Request a DMAMUX channel by static allocate mechanism */
channel = kDMAMGR_STATIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
;
```

38.3.2 DMAMGR dynamic channel allocation

```
uint8_t channel;
dmamanager_handle_t dmamanager_handle;

/* Initialize DMAMGR */
DMAMGR_Init(&dmamanager_handle, EXAMPLE_DMA_BASEADDR, DMA_CHANNEL_NUMBER, startChannel);
/* Request a DMAMUX channel by Dynamic allocate mechanism */
channel = DMAMGR_DYNAMIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
;
```

Data Structures

- struct `dmamanager_handle_t`
dmamanager handle typedef. [More...](#)

Data Structure Documentation

Macros

- #define **DMAMGR_DYNAMIC_ALLOCATE** 0xFFU
Dynamic channel allocation mechanism.

Enumerations

- enum **_dma_manager_status** {
 kStatus_DMAMGR_ChannelOccupied = MAKE_STATUS(kStatusGroup_DMAMGR, 0),
 kStatus_DMAMGR_ChannelNotUsed = MAKE_STATUS(kStatusGroup_DMAMGR, 1),
 kStatus_DMAMGR_NoFreeChannel = MAKE_STATUS(kStatusGroup_DMAMGR, 2) }
DMA manager status.

DMAMGR Initialization and De-initialization

- void **DMAMGR_Init** (**dmamanager_handle_t** *dmamanager_handle, **DMA_Type** *dma_base, **uint32_t** channelNum, **uint32_t** startChannel)
Initializes the DMA manager.
- void **DMAMGR_Deinit** (**dmamanager_handle_t** *dmamanager_handle)
Deinitializes the DMA manager.

DMAMGR Operation

- **status_t DMAMGR_RequestChannel** (**dmamanager_handle_t** *dmamanager_handle, **uint32_t** requestSource, **uint32_t** channel, **void** *handle)
Requests a DMA channel.
- **status_t DMAMGR_ReleaseChannel** (**dmamanager_handle_t** *dmamanager_handle, **void** *handle)
Releases a DMA channel.
- **bool DMAMGR_IsChannelOccupied** (**dmamanager_handle_t** *dmamanager_handle, **uint32_t** channel)
Get a DMA channel status.

38.4 Data Structure Documentation

38.4.1 struct dmamanager_handle_t

Note

The contents of this structure are private and subject to change.

This dma manager handle structure is used to store the parameters transferred by users. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.

Data Fields

- **void *dma_base**
Peripheral DMA instance.
- **uint32_t channelNum**

- *Channel numbers for the DMA instance which need to be managed by dma manager.*
- `uint32_t startChannel`
The start channel that can be managed by dma manager; users need to transfer it with a certain number or NULL.
- `bool s_DMAMGR_Channels [64]`
The s_DMAMGR_Channels is used to store dma manager state.
- `uint32_t DmamuxInstanceStart`
The DmamuxInstance is used to calculate the DMAMUX Instance according to the DMA Instance.
- `uint32_t multiple`
The multiple is used to calculate the multiple between DMAMUX count and DMA count.

38.4.1.0.0.79 Field Documentation

38.4.1.0.0.79.1 `void* dmamanager_handle_t::dma_base`

38.4.1.0.0.79.2 `uint32_t dmamanager_handle_t::channelNum`

38.4.1.0.0.79.3 `uint32_t dmamanager_handle_t::startChannel`

38.4.1.0.0.79.4 `bool dmamanager_handle_t::s_DMAMGR_Channels[64]`

38.4.1.0.0.79.5 `uint32_t dmamanager_handle_t::DmamuxInstanceStart`

38.4.1.0.0.79.6 `uint32_t dmamanager_handle_t::multiple`

38.5 Macro Definition Documentation

38.5.1 `#define DMAMGR_DYNAMIC_ALLOCATE 0xFFU`

38.6 Enumeration Type Documentation

38.6.1 `enum _dma_manager_status`

Enumerator

`kStatus_DMAMGR_ChannelOccupied` Channel has been occupied.

`kStatus_DMAMGR_ChannelNotUsed` Channel has not been used.

`kStatus_DMAMGR_NoFreeChannel` All channels have been occupied.

38.7 Function Documentation

38.7.1 `void DMAMGR_Init (dmamanager_handle_t * dmamanager_handle, DMA_Type * dma_base, uint32_t channelNum, uint32_t startChannel)`

This function initializes the DMA manager, ungates the DMAMUX clocks, and initializes the eDMA or DMA peripherals.

Function Documentation

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>dma_base</i>	Peripheral DMA instance base pointer.
<i>dmamux_base</i>	Peripheral DMAMUX instance base pointer.
<i>channelNum</i>	Channel numbers for the DMA instance which need to be managed by dma manager.
<i>startChannel</i>	The start channel that can be managed by dma manager.

38.7.2 void DMAMGR_Deinit (*dmamanager_handle_t * dmamanager_handle*)

This function deinitializes the DMA manager, disables the DMAMUX channels, gates the DMAMUX clocks, and deinitializes the eDMA or DMA peripherals.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
--------------------------	---

38.7.3 status_t DMAMGR_RequestChannel (*dmamanager_handle_t * dmamanager_handle, uint32_t requestSource, uint32_t channel, void * handle*)

This function requests a DMA channel which is not occupied. The two channels to allocate the mechanism are dynamic and static channels. For the dynamic allocation mechanism (channe = DMAMGR_DYNAMIC_ALLOCATE), DMAMGR allocates a DMA channel according to the given request source and start-Channel and then configures it. For static allocation mechanism, DMAMGR configures the given channel according to the given request source and channel number.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>requestSource</i>	DMA channel request source number. See the soc.h, see the enum <code>dma_request_source_t</code>
<i>channel</i>	The channel number users want to occupy. If using the dynamic channel allocate mechanism, set the channel equal to <code>DMAMGR_DYNAMIC_ALLOCATE</code> .
<i>handle</i>	DMA or eDMA handle pointer.

Return values

<i>kStatus_Success</i>	In a dynamic/static channel allocation mechanism, allocate the DMAMUX channel successfully.
<i>kStatus_DMAMGR_No_FreeChannel</i>	In a dynamic channel allocation mechanism, all DMAMUX channels are occupied.
<i>kStatus_DMAMGR_ChannelOccupied</i>	In a static channel allocation mechanism, the given channel is occupied.

38.7.4 **status_t DMAMGR_ReleaseChannel (dmamanager_handle_t * dmamanager_handle, void * handle)**

This function releases an occupied DMA channel.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>handle</i>	DMA or eDMA handle pointer.

Return values

Function Documentation

<i>kStatus_Success</i>	Releases the given channel successfully.
<i>kStatus_DMAMGR_-ChannelNotUsed</i>	The given channel to be released had not been used before.

38.7.5 **bool DMAMGR_IsChannelOccupied (dmamanager_handle_t * dmamanager_handle, uint32_t channel)**

This function get a DMA channel status. Return 0 indicates the channel has not been used, return 1 indicates the channel has been occupied.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>channel</i>	The channel number that users want get its status.

Chapter 39

Memory-Mapped Cryptographic Acceleration Unit (MMCAU)

39.1 Overview

The mmCAU software library uses the mmCAU co-processor that is connected to the ARM Cortex-M4-/M0+ Private Peripheral Bus (PPB). In this chapter, CAU refers to both CAU and mmCAU unless explicitly noted.

39.2 Purpose

The following chapter describes how to use the mmCAU software library in any application to integrate a cryptographic algorithm or hashing function supported by the software library. NXP products supported by the software library are MCU/MPUs. Check the specific Freescale product for CAU availability.

39.3 Library Features

The library is as compact and generic as possible to simplify the integration with existing cryptographic software. The library has a standard header file with ANSI C prototypes for all functions: "cau_api.h". This software library is thread safe only if CAU registers are saved on a context switch. The mmCAU software library is also compatible to ARM C compiler conventions (EABI). All pointers passed to mmCAU API functions (input and output data blocks, keys, key schedules, and so on) are aligned to 0-modulo-4 addresses.

For applications that don't need to deal with the aligned addresses, a simple wrapper layer is provided. The wrapper layer consists of the "fsl_mmcau.h" header file and "fsl_mmcau.c" source code file. The only function of the wrapper layer is that it supports unaligned addresses

. The CAU library supports the following encryption/decryption algorithms and hashing functions:

- AES128
- AES192
- AES256
- DES
- MD5
- SHA1
- SHA256

Note: 3DES crypto algorithms are supported by calling the corresponding DES crypto function three times. Hardware support for SHA256 is only present in the CAU version 2. See the appropriate MC-U/MPU reference manual for details about availability. Additionally, the [cau_sha256_initialize_output\(\)](#) function checks the hardware revision and returns a (-1) value if the CAU lacks SHA256 support.

39.4 CAU and mmCAU software library overview

Table 1 shows the crypto algorithms and hashing functions included in the software library:

mmCAU software library usage

Crypto Algorithms	AES128 AES192 AES256	cau_aes_set_key
		cau_aes_encrypt
		cau_aes_decrypt
	DES/3DES	cau_des_chk_parity
		cau_des_encrypt
		cau_des_decrypt
Hashing Functions	MD5	cau_md5_initialize_output
		cau_md5_hash_n
		cau_md5_update
		cau_md5_hash
	SHA1	cau_sha1_initialize_output
		cau_sha1_hash_n
		cau_sha1_update
		cau_sha1_hash
	SHA256	cau_sha256_initialize_output
		cau_sha256_hash_n
		cau_sha256_update
		cau_sha256_hash

Table 1: Library Overview

39.5 mmCAU software library usage

The software library contains the following files:

File	Description
cau_api.h	CAU and mmCAU header file
lib_mmcau.a	mmCAU library

Table 2: File Description

The header file and lib_mmcau.a must always be included in the project.

Functions

- void `cau_aes_set_key` (const unsigned char *key, const int key_size, unsigned char *key_sch)
AES: Performs an AES key expansion.
- void `cau_aes_encrypt` (const unsigned char *in, const unsigned char *key_sch, const int nr, unsigned char *out)

- void [cau_aes_decrypt](#) (const unsigned char *in, const unsigned char *key_sch, const int nr, unsigned char *out)

AES: Encrypts a single 16 byte block.
- int [cau_des_chk_parity](#) (const unsigned char *key)

DES: Checks key parity.
- void [cau_des_encrypt](#) (const unsigned char *in, const unsigned char *key, unsigned char *out)

DES: Encrypts a single 8-byte block.
- void [cau_des_decrypt](#) (const unsigned char *in, const unsigned char *key, unsigned char *out)

DES: Decrypts a single 8-byte block.
- void [cau_md5_initialize_output](#) (const unsigned char *md5_state)

MD5: Initializes the MD5 state variables.
- void [cau_md5_hash_n](#) (const unsigned char *msg_data, const int num_blks, unsigned char *md5_state)

MD5: Updates MD5 state variables with n message blocks.
- void [cau_md5_update](#) (const unsigned char *msg_data, const int num_blks, unsigned char *md5_state)

MD5: Updates MD5 state variables.
- void [cau_md5_hash](#) (const unsigned char *msg_data, unsigned char *md5_state)

MD5: Updates MD5 state variables with one message block.
- void [cau_sha1_initialize_output](#) (const unsigned int *sha1_state)

SHA1: Initializes the SHA1 state variables.
- void [cau_sha1_hash_n](#) (const unsigned char *msg_data, const int num_blks, unsigned int *sha1_state)

SHA1: Updates SHA1 state variables with n message blocks.
- void [cau_sha1_update](#) (const unsigned char *msg_data, const int num_blks, unsigned int *sha1_state)

SHA1: Updates SHA1 state variables.
- void [cau_sha1_hash](#) (const unsigned char *msg_data, unsigned int *sha1_state)

SHA1: Updates SHA1 state variables with one message block.
- int [cau_sha256_initialize_output](#) (const unsigned int *output)

SHA256: Initializes the SHA256 state variables.
- void [cau_sha256_hash_n](#) (const unsigned char *input, const int num_blks, unsigned int *output)

SHA256: Updates SHA256 state variables with n message blocks.
- void [cau_sha256_update](#) (const unsigned char *input, const int num_blks, unsigned int *output)

SHA256: Updates SHA256 state variables.
- void [cau_sha256_hash](#) (const unsigned char *input, unsigned int *output)

SHA256: Updates SHA256 state variables with one message block.
- status_t [MMCAU_AES_SetKey](#) (const uint8_t *key, const size_t keySize, uint8_t *keySch)

AES: Performs an AES key expansion.
- status_t [MMCAU_AES_EncryptEcb](#) (const uint8_t *in, const uint8_t *keySch, uint32_t aesRounds, uint8_t *out)

AES: Encrypts a single 16 byte block.
- status_t [MMCAU_AES_DecryptEcb](#) (const uint8_t *in, const uint8_t *keySch, uint32_t aesRounds, uint8_t *out)

AES: Decrypts a single 16-byte block.
- status_t [MMCAU_DES_ChkParity](#) (const uint8_t *key)

DES: Checks the key parity.
- status_t [MMCAU_DES_EncryptEcb](#) (const uint8_t *in, const uint8_t *key, uint8_t *out)

DES: Encrypts a single 8-byte block.

Function Documentation

- status_t [MMCAU_DES_DecryptEcb](#) (const uint8_t *in, const uint8_t *key, uint8_t *out)
DES: Decrypts a single 8-byte block.
- status_t [MMCAU_MD5_InitializeOutput](#) (uint32_t *md5State)
MD5: Initializes the MD5 state variables.
- status_t [MMCAU_MD5_HashN](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *md5State)
MD5: Updates the MD5 state variables with n message blocks.
- status_t [MMCAU_MD5_Update](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *md5State)
MD5: Updates the MD5 state variables.
- status_t [MMCAU_SHA1_InitializeOutput](#) (uint32_t *sha1State)
SHA1: Initializes the SHA1 state variables.
- status_t [MMCAU_SHA1_HashN](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *sha1State)
SHA1: Updates the SHA1 state variables with n message blocks.
- status_t [MMCAU_SHA1_Update](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *sha1State)
SHA1: Updates the SHA1 state variables.
- status_t [MMCAU_SHA256_InitializeOutput](#) (uint32_t *sha256State)
SHA256: Initializes the SHA256 state variables.
- status_t [MMCAU_SHA256_HashN](#) (const uint8_t *input, uint32_t numBlocks, uint32_t *sha256State)
SHA256: Updates the SHA256 state variables with n message blocks.
- status_t [MMCAU_SHA256_Update](#) (const uint8_t *input, uint32_t numBlocks, uint32_t *sha256State)
SHA256: Updates SHA256 state variables.

39.6 Function Documentation

39.6.1 void cau_aes_set_key (const unsigned char * key, const int key_size, unsigned char * key_sch)

This function performs an AES key expansion

Parameters

	<i>key</i>	Pointer to input key (128, 192, 256 bits in length).
	<i>key_size</i>	Key size in bits (128, 192, 256)
out	<i>key_sch</i>	Pointer to key schedule output (44, 52, 60 longwords)

Note

All pointers must have word (4 bytes) alignment

Table below shows the requirements for the [cau_aes_set_key\(\)](#) function when using AES128, AES192 or AES256.

[in] Key Size (bits)	[out] Key Schedule Size (32 bit data values)
128	44
192	52
256	60

39.6.2 void cau_aes_encrypt(const unsigned char * *in*, const unsigned char * *key_sch*, const int *nr*, unsigned char * *out*)

This function encrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

	<i>in</i>	Pointer to 16-byte block of input plaintext
	<i>key_sch</i>	Pointer to key schedule (44, 52, 60 longwords)
	<i>nr</i>	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	<i>out</i>	Pointer to 16-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

Table below shows the requirements for the [cau_aes_encrypt\(\)](#)/[cau_aes_decrypt\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
AES128	44	10
AES192	52	12
AES256	60	14

39.6.3 void cau_aes_decrypt(const unsigned char * *in*, const unsigned char * *key_sch*, const int *nr*, unsigned char * *out*)

This function decrypts a single 16-byte block for AES128, AES192 and AES256

Function Documentation

Parameters

	<i>in</i>	Pointer to 16-byte block of input ciphertext
	<i>key_sch</i>	Pointer to key schedule (44, 52, 60 longwords)
	<i>nr</i>	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	<i>out</i>	Pointer to 16-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

Table below shows the requirements for the [cau_aes_encrypt\(\)](#)/[cau_aes_decrypt\(\)](#) function when using AES128, AES192 or AES256.

	Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
	AES128	44	10
	AES192	52	12
	AES256	60	14

39.6.4 int cau_des_chk_parity (const unsigned char * key)

This function checks the parity of a DES key

Parameters

<i>key</i>	64-bit DES key with parity bits. Must have word (4 bytes) alignment.
------------	--

Returns

0 no error

-1 parity error

39.6.5 void cau_des_encrypt (const unsigned char * *in*, const unsigned char * *key*, unsigned char * *out*)

This function encrypts a single 8-byte block with DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input plaintext
	<i>key</i>	Pointer to 64-bit DES key with parity bits
out	<i>out</i>	Pointer to 8-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

39.6.6 void cau_des_decrypt (const unsigned char * *in*, const unsigned char * *key*, unsigned char * *out*)

This function decrypts a single 8-byte block with DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input ciphertext
	<i>key</i>	Pointer to 64-bit DES key with parity bits
out	<i>out</i>	Pointer to 8-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

39.6.7 void cau_md5_initialize_output (const unsigned char * *md5_state*)

This function initializes the MD5 state variables. The output can be used as input to [cau_md5_hash\(\)](#) and [cau_md5_hash_n\(\)](#).

Parameters

out	<i>md5_state</i>	Pointer to 128-bit block of md5 state variables: a,b,c,d
-----	------------------	--

Note

All pointers must have word (4 bytes) alignment

Function Documentation

39.6.8 void cau_md5_hash_n (const unsigned char * *msg_data*, const int *num_blk*s, unsigned char * *md5_state*)

This function updates MD5 state variables for one or more input message blocks

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
in, out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_md5_initialize_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

39.6.9 void cau_md5_update (const unsigned char * *msg_data*, const int *num_blk*s, unsigned char * *md5_state*)

This function updates MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_md5_initialize_output\(\)](#) first.

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_md5_initialize_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

39.6.10 void cau_md5_hash (const unsigned char * *msg_data*, unsigned char * *md5_state*)

This function updates MD5 state variables for one input message block

Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in,out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_md5_initialize_output\(\)](#) function must be called when starting a new hash.

39.6.11 void cau_sha1_initialize_output (const unsigned int * *sha1_state*)

This function initializes the SHA1 state variables. The output can be used as input to [cau_sha1_hash\(\)](#) and [cau_sha1_hash_n\(\)](#).

Parameters

out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e
-----	-------------------	---

Note

All pointers must have word (4 bytes) alignment

39.6.12 void cau_sha1_hash_n (const unsigned char * *msg_data*, const int *num_blk*s, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one or more input message blocks

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
in,out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha1_initialize_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

39.6.13 void cau_sha1_update (const unsigned char * *msg_data*, const int *num_blk*s, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_sha1_initialize_output\(\)](#) first.

Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha1_initialize_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

39.6.14 void cau_sha1_hash (const unsigned char * *msg_data*, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one input message block

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in, out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha1_initialize_output\(\)](#) function must be called when starting a new hash.

39.6.15 int cau_sha256_initialize_output (const unsigned int * *output*)

This function initializes the SHA256 state variables. The output can be used as input to [cau_sha256_hash\(\)](#) and [cau_sha256_hash_n\(\)](#).

Parameters

out	<i>output</i>	Pointer to 256-bit block of SHA2 state variables a,b,c,d,e,f,g,h
-----	---------------	--

Note

All pointers must have word (4 bytes) alignment

Returns

- 0 No error. CAU hardware support for SHA256 is present.
- 1 Error. CAU hardware support for SHA256 is not present.

39.6.16 void cau_sha256_hash_n (const unsigned char * *input*, const int *num_blk*s, unsigned int * *output*)

This function updates SHA256 state variables for one or more input message blocks

Parameters

	<i>input</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
in, out	<i>output</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha256_initialize_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

39.6.17 void cau_sha256_update (const unsigned char * *input*, const int *num_blk*s, unsigned int * *output*)

This function updates SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_sha256_initialize_output\(\)](#) first.

Function Documentation

Parameters

	<i>input</i>	Pointer to start of input message data
	<i>num_blk</i>	Number of 512-bit blocks to process
out	<i>output</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha256_initialize_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

39.6.18 void cau_sha256_hash (const unsigned char * *input*, unsigned int * *output*)

This function updates SHA256 state variables for one input message block

Parameters

	<i>input</i>	Pointer to start of 512-bits of input message data
in, out	<i>output</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha256_initialize_output\(\)](#) function must be called when starting a new hash.

39.6.19 status_t MMCAU_AES_SetKey (const uint8_t * *key*, const size_t *keySize*, uint8_t * *keySch*)

This function performs an AES key expansion.

Parameters

	<i>key</i>	Pointer to input key (128, 192, 256 bits in length).
	<i>keySize</i>	Key size in bytes (16, 24, 32)
out	<i>keySch</i>	Pointer to key schedule output (44, 52, 60 longwords)

Note

Table below shows the requirements for the [MMCAU_AES_SetKey\(\)](#) function when using AES128, AES192, or AES256.

	[in] Key Size (bits)	[out] Key Schedule Size (32 bit data values)
	128	44
	192	52
	256	60

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

39.6.20 **status_t MMCAU_AES_EncryptEcb (const uint8_t * *in*, const uint8_t * *keySch*, uint32_t *aesRounds*, uint8_t * *out*)**

This function encrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	<i>in</i>	Pointer to 16-byte block of input plaintext.
	<i>keySch</i>	Pointer to key schedule (44, 52, 60 longwords).
	<i>aesRounds</i>	Number of AES rounds (10, 12, 14 = f(key_schedule)).
out	<i>out</i>	Pointer to 16-byte block of output ciphertext.

Note

Input and output blocks may overlap.

Table below shows the requirements for the [MMCAU_AES_EncryptEcb\(\)](#)/[MMCAU_AES-DecryptEcb\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
AES128	44	10
AES192	52	12
AES256	60	14

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

Function Documentation

39.6.21 status_t MMCAU_AES_DecryptEcb (const uint8_t * *in*, const uint8_t * *keySch*, uint32_t *aesRounds*, uint8_t * *out*)

This function decrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	<i>in</i>	Pointer to 16-byte block of input ciphertext.
	<i>keySch</i>	Pointer to key schedule (44, 52, 60 longwords).
	<i>aesRounds</i>	Number of AES rounds (10, 12, 14 = f(key_schedule)).
<i>out</i>	<i>out</i>	Pointer to 16-byte block of output plaintext.

Note

Input and output blocks may overlap.

Table below shows the requirements for the [cau_aes_encrypt\(\)](#)/[cau_aes_decrypt\(\)](#). function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
AES128	44	10
AES192	52	12
AES256	60	14

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

39.6.22 status_t MMCAU_DES_ChkParity (const uint8_t * key)

This function checks the parity of a DES key.

Parameters

<i>key</i>	64-bit DES key with parity bits.
------------	----------------------------------

Returns

kStatus_Success No error.

kStatus_Fail Parity error.

kStatus_InvalidArgument Key argument is NULL.

39.6.23 status_t MMCAU_DES_EncryptEcb (const uint8_t * in, const uint8_t * key, uint8_t * out)

This function encrypts a single 8-byte block with the DES algorithm.

Function Documentation

Parameters

	<i>in</i>	Pointer to 8-byte block of input plaintext.
	<i>key</i>	Pointer to 64-bit DES key with parity bits.
out	<i>out</i>	Pointer to 8-byte block of output ciphertext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

39.6.24 status_t MMCAU_DES_DecryptEcb (const uint8_t * *in*, const uint8_t * *key*, uint8_t * *out*)

This function decrypts a single 8-byte block with the DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input ciphertext.
	<i>key</i>	Pointer to 64-bit DES key with parity bits.
out	<i>out</i>	Pointer to 8-byte block of output plaintext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

39.6.25 status_t MMCAU_MD5_InitializeOutput (uint32_t * *md5State*)

This function initializes the MD5 state variables. The output can be used as input to [MMCAU_MD5_HashN\(\)](#).

Parameters

out	<i>md5State</i>	Pointer to 128-bit block of md5 state variables: a,b,c,d
-----	-----------------	--

39.6.26 status_t MMCAU_MD5_HashN (*const uint8_t * msgData, uint32_t numBlocks, uint32_t * md5State*)

This function updates the MD5 state variables for one or more input message blocks.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
in,out	<i>md5State</i>	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The [MMCAU_MD5_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

39.6.27 status_t MMCAU_MD5_Update (*const uint8_t * msgData, uint32_t numBlocks, uint32_t * md5State*)

This function updates the MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls [MMCAU_MD5_InitializeOutput\(\)](#) first.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
out	<i>md5State</i>	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The [MMCAU_MD5_InitializeOutput\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

Function Documentation

39.6.28 **status_t MMCAU_SHA1_InitializeOutput(uint32_t * sha1State)**

This function initializes the SHA1 state variables. The output can be used as input to [MMCAU_SHA1_HashN\(\)](#).

Parameters

<code>out</code>	<code>sha1State</code>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.
------------------	------------------------	--

39.6.29 `status_t MMCAU_SHA1_HashN(const uint8_t * msgData, uint32_t numBlocks, uint32_t * sha1State)`

This function updates the SHA1 state variables for one or more input message blocks.

Parameters

	<code>msgData</code>	Pointer to start of input message data.
	<code>numBlocks</code>	Number of 512-bit blocks to process.
<code>in, out</code>	<code>sha1State</code>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA1_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

39.6.30 `status_t MMCAU_SHA1_Update(const uint8_t * msgData, uint32_t numBlocks, uint32_t * sha1State)`

This function updates the SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls [MMCAU_SHA1_InitializeOutput\(\)](#) first.

Parameters

	<code>msgData</code>	Pointer to start of input message data.
	<code>numBlocks</code>	Number of 512-bit blocks to process.
<code>out</code>	<code>sha1State</code>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA1_InitializeOutput\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

Function Documentation

39.6.31 **status_t MMCAU_SHA256_InitializeOutput (uint32_t * sha256State)**

This function initializes the SHA256 state variables. The output can be used as input to [MMCAU_SHA256_HashN\(\)](#).

Parameters

<code>out</code>	<code>sha256State</code>	Pointer to 256-bit block of SHA2 state variables a, b, c, d, e, f, g, h.
------------------	--------------------------	--

Returns

`kStatus_Success` No error. CAU hardware support for SHA256 is present.

`kStatus_Fail` Error. CAU hardware support for SHA256 is not present.

`kStatus_InvalidArgument` Error. `sha256State` is NULL.

39.6.32 `status_t MMCAU_SHA256_HashN (const uint8_t * input, uint32_t numBlocks, uint32_t * sha256State)`

This function updates SHA256 state variables for one or more input message blocks.

Parameters

	<code>input</code>	Pointer to start of input message data.
	<code>numBlocks</code>	Number of 512-bit blocks to process.
<code>in, out</code>	<code>sha256State</code>	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA256_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

39.6.33 `status_t MMCAU_SHA256_Update (const uint8_t * input, uint32_t numBlocks, uint32_t * sha256State)`

This function updates the SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_sha256_initialize_output\(\)](#) first.

Parameters

	<code>input</code>	Pointer to start of input message data.
	<code>numBlocks</code>	Number of 512-bit blocks to process.
<code>out</code>	<code>sha256State</code>	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

Function Documentation

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA256_InitializeOutput\(\)](#) function is not required to be called, as it is called internally to start a new hash. All input message blocks must be contiguous.

Chapter 40

Secure Digital Card/Embedded MultiMedia Card (CARD)

40.1 Overview

The MCUXpresso SDK provides a driver to access the Secure Digital Card and Embedded MultiMedia Card based on the SDHC/USDHC/SDIF driver.

Function groups

This function group implements the SD card functional API.

This function group implements the MMC card functional API.

Typical use case

40.2 SD CARD Operation

error log support

Lots of error log has been added to sd relate functions, if error occurs during initial/read/write, please enable the error log print functionality with #define SDMMC_ENABLE_LOG_PRINT 1 And rerun the project then user can check what kind of error happened.

User configurable

```
typedef struct _sd_card
{
    SDMMCHOST_CONFIG host;
    sdcards_usr_param_t usrParam;
    bool isHostReady;
    bool noInternalAlign;
    uint32_t busClock_Hz;
    uint32_t relativeAddress;
    uint32_t version;
    uint32_t flags;
    uint32_t rawCid[4U];
    uint32_t rawCsd[4U];
    uint32_t rawScr[2U];
    uint32_t ocr;
    sd_cid_t cid;
    sd_csd_t csd;
    sd_scr_t scr;
    uint32_t blockCount;
    uint32_t blockSize;
    sd_timing_mode_t currentTiming;
    sd_driver_strength_t driverStrength;
    sd_max_current_t maxCurrent;
    sdmmc_operation_voltage_t operationVoltage;
} sd_card_t;
```

SD CARD Operation

Part of The variables above is user configurable,

1. SDMMCHOST_CONFIG host Application need to provide host controller base address and the host's source clock frequency.
2. `sdcards_usr_param_t` usrParam Two member in the userParam structure, a. cd-which allow application define the card insert/remove callback function, redefine the card detect timeout ms and also allow application determine how to detect card. b. pwr-which allow application redefine the power on/off function and the power on/off delay ms. However, sdmmc always use the default setting if application not define it in application. The default setting is depend on the macro defined in the board.h.
3. bool noInternalAlign Sdmmc include an address align internal buffer(to use host controller internal DMA), to improve read/write performance while application cannot make sure the data address used to read/write is align, set it to true will achieve a better performance.
4. `sd_timing_mode_t` currentTiming It is used to indicate the currentTiming the card is working on, however sdmmc also support preset timing mode, then sdmmc will try to switch to this timing first, if failed, a valid timing will switch to automatically. Generally, user may not set this variable if you don't know what kind of timing the card support, sdmmc will switch to the highest timing which the card support.
5. `sd_driver_strength_t` driverStrength Choose a valid card driver strength if application required and call `SD_SetDriverStrength` in application.
6. `sd_max_current_t` maxCurrent Choose a valid card current if application required and call `SD_SetMaxCurrent` in application.

Board dependency

Sdmmc depend on some board specific settings, such as card detect - which is used to detect card, the card active level is determine by the socket you are using, low level is active usually, but some socket use high as active, please set the macro `#define BOARD_xxxx_CARD_INSERT_CD_LEVEL (0U)` According to your board specific if you cannot detect card even if card is inserted.

power control - Macro for USDHC only, SDHC/SDIF support high speed timing only, so please ignore the power reset pin for SDHC/SDIF. which is used to reset card power for UHS card, to make UHS timing work properly, please make sure the power reset pin is configured properly in board.h. `#define BOARD_SD_POWER_RESET_GPIO (GPIO1) #define BOARD_SD_POWER_RESET_GPIO_PIN (5U) #define BOARD_USDHC_SDCARD_POWER_CONTROL_INIT()`

pin configurations - Function for USDHC only. which is used to switch the signal pin configurations include driver strength/speed mode dynamically for different timing mode, reference the function defined in board.c void `BOARD_SD_Pin_Config(uint32_t speed, uint32_t strength)`

Typical use case

```
/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
```

```

/*Redefine the cd and pwr in the application if required*/
card->usrParam.cd = &s_sdCardDetect;
card->usrParam.pwr = &s_sdCardPwrCtrl;
/* intial the host controller */
SD_HostInit(card);
/*wait card inserted, before detect card you can power off card first and power on again after card
   inserted, such as*/
SD_PowerOffCard(card->host.base, card->usrParam.pwr);
SD_WaitCardDetectStatus(card->host.base, card->usrParam.cd, true);
SD_PowerOnCard(card->host.base, card->usrParam.pwr);
/*call card initial function*/
SD_CardInit(card);

/* Or you can call below function directly, it is a highlevel init function which will include host
   initialize, card detect, card initial */

/* Init card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

/* after initialization finised, access the card with below functions. */

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);

```

40.3 MMC CARD Operation

error log support

Not support yet

User configurable

Board dependency

Typical use case

```
/* Save host information. */
```

SDIO CARD Operation

```
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
/* MMC card VCC supply, only allow 3.3 or 1.8v, depend on your board config.
 * If a power reset circuit is available on you board for mmc, and 1.8v is supported,
 * #define BOARD_USDHC_MMCCARD_POWER_CONTROL_INIT()
 * #define BOARD_USDHC_MMCCARD_POWER_CONTROL(state)
 * in board.h must be implemented.
 * User can remove preset the voltage window and sdmmc will switch VCC automatically. */
card->hostVoltageWindowVCC = BOARD_MMC_VCC_SUPPLY;

/* Init card.*/
if (MMC_Init(card))
{
    PRINTF ("\n MMC card init failed \n");
}

/* after initialization finised, access the card with below functions. */
while (true)
{
    if (kStatus_Success != MMC_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF ("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != MMC_ReadBlocks(card, g_dataRead, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF ("Read multiple data blocks failed.\r\n");
    }
}
MMC_Deinit(card);
```

40.4 SDIO CARD Operation

error log support

Not support yet

User configurable

Board dependency

Typical use case

Data Structures

- struct [sdmmchost_detect_card_t](#)
sd card detect [More...](#)
- struct [sdmmchost_pwr_card_t](#)
card power control [More...](#)

Macros

- #define [SDMMCHOST_NOT_SUPPORT](#) 0U

- use this define to indicate the host not support feature

- #define **SDMMCHOST_SUPPORT** 1U

use this define to indicate the host support feature

- #define **kSDMMCHOST_DATABUSWIDTH1BIT** kSDHC_DataBusWidth1Bit

1-bit mode

- #define **kSDMMCHOST_DATABUSWIDTH4BIT** kSDHC_DataBusWidth4Bit

4-bit mode

- #define **kSDMMCHOST_DATABUSWIDTH8BIT** kSDHC_DataBusWidth8Bit

8-bit mode

- #define **SDMMCHOST_STANDARD_TUNING_START** (0U)

standard tuning start point

- #define **SDMMCHOST_TUINIG_STEP** (1U)

standard tuning step

- #define **SDMMCHOST_RETUNING_TIMER_COUNT** (4U)

Re-tuning timer.

Typedefs

- typedef void(* **sdmmchost_cd_callback_t**)(bool isInserted, void *userData)
card detect callback definition
- typedef void(* **sdmmchost_pwr_t**)(void)
card power control function pointer

Enumerations

- enum **_host_capability**
SDHC host capability.
- enum **_sdmmchost_endian_mode** {
 kSDMMCHOST_EndianModeBig = 0U,
 kSDMMCHOST_EndianModeHalfWordBig = 1U,
 kSDMMCHOST_EndianModeLittle = 2U }
host Endian mode corresponding to driver define
- enum **sdmmchost_detect_card_type_t** {
 kSDMMCHOST_DetectCardByGpioCD,
 kSDMMCHOST_DetectCardByHostCD,
 kSDMMCHOST_DetectCardByHostDATA3 }
sd card detect type

Variables

- **sdmmchost_detect_card_type_t** sdmmchost_detect_card_t::cdType
card detect type
- **uint32_t** sdmmchost_detect_card_t::cdTimeOut_ms
card detect timeout which allow 0 - 0xFFFFFFFF, value 0 will return immediately. 0xFFFFFFFF will block until card is insert
- **sdmmchost_cd_callback_t** sdmmchost_detect_card_t::cardInserted
card inserted callback which is meaningful for interrupt case
- **sdmmchost_cd_callback_t** sdmmchost_detect_card_t::cardRemoved
card removed callback which is meaningful for interrupt case
- **void *** sdmmchost_detect_card_t::userData

Data Structure Documentation

- *user data*
- `sdmmchost_pwr_t` `sdmmchost_pwr_card_t::powerOn`
power on function pointer
- `uint32_t` `sdmmchost_pwr_card_t::powerOnDelay_ms`
power on delay
- `sdmmchost_pwr_t` `sdmmchost_pwr_card_t::powerOff`
power off function pointer
- `uint32_t` `sdmmchost_pwr_card_t::powerOffDelay_ms`
power off delay

adaptor function

- static `status_t` `SDMMCHOST_NotSupport` (`void *parameter`)
host not support function, this function is used for host not support feature
- `status_t` `SDMMCHOST_WaitCardDetectStatus` (`SDMMCHOST_TYPE *hostBase, const sdmmchost_detect_card_t *cd, bool waitCardStatus`)
Detect card insert, only need for SD cases.
- `bool` `SDMMCHOST_IsCardPresent` (`void`)
check card is present or not.
- `status_t` `SDMMCHOST_Init` (`SDMMCHOST_CONFIG *host, void *userData`)
Init host controller.
- `void` `SDMMCHOST_Reset` (`SDMMCHOST_TYPE *base`)
reset host controller.
- `void` `SDMMCHOST_ErrorRecovery` (`SDMMCHOST_TYPE *base`)
host controller error recovery.
- `void` `SDMMCHOST_Deinit` (`void *host`)
Deinit host controller.
- `void` `SDMMCHOST_PowerOffCard` (`SDMMCHOST_TYPE *base, const sdmmchost_pwr_card_t *pwr`)
host power off card function.
- `void` `SDMMCHOST_PowerOnCard` (`SDMMCHOST_TYPE *base, const sdmmchost_pwr_card_t *pwr`)
host power on card function.
- `void` `SDMMCHOST_Delay` (`uint32_t milliseconds`)
SDMMC host delay function.

40.5 Data Structure Documentation

40.5.1 struct `sdmmchost_detect_card_t`

Data Fields

- `sdmmchost_detect_card_type_t cdType`
card detect type
- `uint32_t cdTimeOut_ms`
card detect timeout which allow 0 - 0xFFFFFFFF, value 0 will return immediately. 0xFFFFFFFF will block until card is insert
- `sdmmchost_cd_callback_t cardInserted`
card inserted callback which is meaningful for interrupt case
- `sdmmchost_cd_callback_t cardRemoved`

- *card removed callback which is meaningful for interrupt case*
 • void * **userData**
user data

40.5.2 struct sdmmchost_pwr_card_t

Data Fields

- **sdmmchost_pwr_t powerOn**
power on function pointer
- **uint32_t powerOnDelay_ms**
power on delay
- **sdmmchost_pwr_t powerOff**
power off function pointer
- **uint32_t powerOffDelay_ms**
power off delay

40.6 Enumeration Type Documentation

40.6.1 enum _sdmmchost_endian_mode

Enumerator

kSDMMCHOST_EndianModeBig Big endian mode.

kSDMMCHOST_EndianModeHalfWordBig Half word big endian mode.

kSDMMCHOST_EndianModeLittle Little endian mode.

40.6.2 enum sdmmchost_detect_card_type_t

Enumerator

kSDMMCHOST_DetectCardByGpioCD sd card detect by CD pin through GPIO

kSDMMCHOST_DetectCardByHostCD sd card detect by CD pin through host

kSDMMCHOST_DetectCardByHostDATA3 sd card detect by DAT3 pin through host

40.7 Function Documentation

40.7.1 static status_t SDMMCHOST_NotSupport (void * *parameter*) [inline], [static]

Function Documentation

Parameters

<i>void</i>	parameter ,used to avoid build warning
-------------	--

Return values

<i>kStatus_Fail,host</i>	do not support
--------------------------	----------------

40.7.2 **status_t SDMMCHOST_WaitCardDetectStatus (SDMMCHOST_TYPE * hostBase, const sdmmchost_detect_card_t * cd, bool waitCardStatus)**

Parameters

<i>base</i>	the pointer to host base address
<i>cd</i>	card detect configuration
<i>waitCardStatus</i>	status which user want to wait

Return values

<i>kStatus_Success</i>	detect card insert
<i>kStatus_Fail</i>	card insert event fail

40.7.3 **bool SDMMCHOST_IsCardPresent (void)**

Return values

<i>true</i>	card is present
<i>false</i>	card is not present

40.7.4 **status_t SDMMCHOST_Init (SDMMCHOST_CONFIG * host, void * userData)**

Parameters

<i>host</i>	the pointer to host structure in card structure.
<i>userData</i>	specific user data

Return values

<i>kStatus_Success</i>	host init success
<i>kStatus_Fail</i>	event fail

40.7.5 void SDMMCHOST_Reset (**SDMMCHOST_TYPE** * *base*)

Parameters

<i>host</i>	base address.
-------------	---------------

40.7.6 void SDMMCHOST_ErrorRecovery (**SDMMCHOST_TYPE** * *base*)

Parameters

<i>host</i>	base address.
-------------	---------------

40.7.7 void SDMMCHOST_Deinit (**void** * *host*)

Parameters

<i>host</i>	the pointer to host structure in card structure.
-------------	--

40.7.8 void SDMMCHOST_PowerOffCard (**SDMMCHOST_TYPE** * *base*, const **sdmmchost_pwr_card_t** * *pwr*)

Parameters

Function Documentation

<i>base</i>	host base address.
<i>pwr</i>	depend on user define power configuration.

40.7.9 void SDMMCHOST_PowerOnCard (SDMMCHOST_TYPE * *base*, const sdmmchost_pwr_card_t * *pwr*)

Parameters

<i>base</i>	host base address.
<i>pwr</i>	depend on user define power configuration.

40.7.10 void SDMMCHOST_Delay (uint32_t *milliseconds*)

Parameters

<i>milliseconds</i>	delay counter.
---------------------	----------------

Chapter 41

SPI based Secure Digital Card (SDSPI)

41.1 Overview

The MCUXpresso SDK provides a driver to access the Secure Digital Card based on the SPI driver.

Function groups

This function group implements the SD card functional API in the SPI mode.

Typical use case

```
/* SPI_Init(). */

/* Register the SDSPI driver callback. */

/* Initializes card. */
if (kStatus_Success != SDSPI_Init(card))
{
    SDSPI_Deinit(card)
    return;
}

/* Read/Write card */
memset(g_testWriteBuffer, 0x17U, sizeof(g_testWriteBuffer));

while (true)
{
    memset(g_testReadBuffer, 0U, sizeof(g_testReadBuffer));

    SDSPI_WriteBlocks(card, g_testWriteBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    SDSPI_ReadBlocks(card, g_testReadBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    if (memcmp(g_testReadBuffer, g_testReadBuffer, sizeof(g_testWriteBuffer)))
    {
        break;
    }
}
```

Data Structures

- struct [sdspi_host_t](#)
SDSPI host state. [More...](#)
- struct [sdspi_card_t](#)
SD Card Structure. [More...](#)

Overview

Enumerations

- enum `_sdspi_status` {
 `kStatus_SDSPI_SetFrequencyFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 0U),
 `kStatus_SDSPI_ExchangeFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 1U),
 `kStatus_SDSPI_WaitReadyFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 2U),
 `kStatus_SDSPI_ResponseError` = MAKE_STATUS(kStatusGroup_SDSPI, 3U),
 `kStatus_SDSPI_WriteProtected` = MAKE_STATUS(kStatusGroup_SDSPI, 4U),
 `kStatus_SDSPI_GoIdleFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 5U),
 `kStatus_SDSPI_SendCommandFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 6U),
 `kStatus_SDSPI_ReadFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 7U),
 `kStatus_SDSPI_WriteFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 8U),
 `kStatus_SDSPI_SendInterfaceConditionFailed`,
 `kStatus_SDSPI_SendOperationConditionFailed`,
 `kStatus_SDSPI_ReadOcrFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 11U),
 `kStatus_SDSPI_SetBlockSizeFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 12U),
 `kStatus_SDSPI_SendCsdFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 13U),
 `kStatus_SDSPI_SendCidFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 14U),
 `kStatus_SDSPI_StopTransmissionFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 15U),
 `kStatus_SDSPI_SendApplicationCommandFailed`,
 `kStatus_SDSPI_InvalidVoltage` = MAKE_STATUS(kStatusGroup_SDSPI, 17U),
 `kStatus_SDSPI_SwitchCmdFail` = MAKE_STATUS(kStatusGroup_SDSPI, 18U),
 `kStatus_SDSPI_NotSupportYet` = MAKE_STATUS(kStatusGroup_SDSPI, 19U) }
 SDSPI API status.
- enum `_sdspi_card_flag` {
 `kSDSPI_SupportHighCapacityFlag` = (1U << 0U),
 `kSDSPI_SupportSdhcFlag` = (1U << 1U),
 `kSDSPI_SupportSdxcFlag` = (1U << 2U),
 `kSDSPI_SupportSdscFlag` = (1U << 3U) }
 SDSPI card flag.
- enum `_sdspi_response_type` {
 `kSDSPI_ResponseTypeR1` = 0U,
 `kSDSPI_ResponseTypeR1b` = 1U,
 `kSDSPI_ResponseTypeR2` = 2U,
 `kSDSPI_ResponseTypeR3` = 3U,
 `kSDSPI_ResponseTypeR7` = 4U }
 SDSPI response type.
- enum `_sdspi_cmd` {
 `kSDSPI_CmdGoIdle` = kSDMMC_GoIdleState << 8U | kSDSPI_ResponseTypeR1,
 `kSDSPI_CmdCrc` = kSDSPI_CommandCrc << 8U | kSDSPI_ResponseTypeR1,
 `kSDSPI_CmdSendInterfaceCondition` }
 SDSPI command type.

SDSPI Function

- `status_t SDSPI_Init (sdspi_card_t *card)`
Initializes the card on a specific SPI instance.

- void **SDSPI_Deinit** (**sdspi_card_t** *card)
Deinitializes the card.
- bool **SDSPI_CheckReadOnly** (**sdspi_card_t** *card)
Checks whether the card is write-protected.
- status_t **SDSPI_ReadBlocks** (**sdspi_card_t** *card, **uint8_t** *buffer, **uint32_t** startBlock, **uint32_t** blockCount)
Reads blocks from the specific card.
- status_t **SDSPI_WriteBlocks** (**sdspi_card_t** *card, **uint8_t** *buffer, **uint32_t** startBlock, **uint32_t** blockCount)
Writes blocks of data to the specific card.
- status_t **SDSPI_SendCid** (**sdspi_card_t** *card)
Send GET-CID command In our sdspi init function, this function is removed for better code size, if id information is needed, you can call it after the init function directly.
- status_t **SDSPI_SendPreErase** (**sdspi_card_t** *card, **uint32_t** blockCount)
Multiple blocks write pre-erase function.
- status_t **SDSPI_EraseBlocks** (**sdspi_card_t** *card, **uint32_t** startBlock, **uint32_t** blockCount)
Block erase function.
- status_t **SDSPI_SwitchToHighSpeed** (**sdspi_card_t** *card)
Switch to high speed function.

41.2 Data Structure Documentation

41.2.1 struct **sdspi_host_t**

Data Fields

- **uint32_t busBaudRate**
Bus baud rate.
- **status_t(* setFrequency)**(**uint32_t** frequency)
Set frequency of SPI.
- **status_t(* exchange)**(**uint8_t** *in, **uint8_t** *out, **uint32_t** size)
Exchange data over SPI.

41.2.2 struct **sdspi_card_t**

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- **sdspi_host_t * host**
Host state information.
- **uint32_t relativeAddress**
Relative address of the card.
- **uint32_t flags**
Flags defined in _sdspi_card_flag.
- **uint8_t rawCid [16U]**
Raw CID content.

Enumeration Type Documentation

- `uint8_t rawCsd [16U]`
Raw CSD content.
- `uint8_t rawScr [8U]`
Raw SCR content.
- `uint32_t ocr`
Raw OCR content.
- `sd_cid_t cid`
CID.
- `sd_csd_t csd`
CSD.
- `sd_scr_t scr`
SCR.
- `uint32_t blockCount`
Card total block number.
- `uint32_t blockSize`
Card block size.

41.2.2.0.0.80 Field Documentation

41.2.2.0.0.80.1 `uint32_t sdspi_card_t::flags`

41.3 Enumeration Type Documentation

41.3.1 `enum _sdspi_status`

Enumerator

- `kStatus_SDSPI_SetFrequencyFailed` Set frequency failed.
- `kStatus_SDSPI_ExchangeFailed` Exchange data on SPI bus failed.
- `kStatus_SDSPI_WaitReadyFailed` Wait card ready failed.
- `kStatus_SDSPI_ResponseError` Response is error.
- `kStatus_SDSPI_WriteProtected` Write protected.
- `kStatus_SDSPI_GoIdleFailed` Go idle failed.
- `kStatus_SDSPI_SendCommandFailed` Send command failed.
- `kStatus_SDSPI_ReadFailed` Read data failed.
- `kStatus_SDSPI_WriteFailed` Write data failed.
- `kStatus_SDSPI_SendInterfaceConditionFailed` Send interface condition failed.
- `kStatus_SDSPI_SendOperationConditionFailed` Send operation condition failed.
- `kStatus_SDSPI_ReadOcrFailed` Read OCR failed.
- `kStatus_SDSPI_SetBlockSizeFailed` Set block size failed.
- `kStatus_SDSPI_SendCsdFailed` Send CSD failed.
- `kStatus_SDSPI_SendCidFailed` Send CID failed.
- `kStatus_SDSPI_StopTransmissionFailed` Stop transmission failed.
- `kStatus_SDSPI_SendApplicationCommandFailed` Send application command failed.
- `kStatus_SDSPI_InvalidVoltage` invalid supply voltage
- `kStatus_SDSPI_SwitchCmdFail` switch command crc protection on/off
- `kStatus_SDSPI_NotSupportYet` not support

41.3.2 enum _sdspi_card_flag

Enumerator

kSDSPI_SupportHighCapacityFlag Card is high capacity.

kSDSPI_SupportSdhcFlag Card is SDHC.

kSDSPI_SupportSdxcFlag Card is SDXC.

kSDSPI_SupportSdscFlag Card is SDSC.

41.3.3 enum _sdspi_response_type

Enumerator

kSDSPI_ResponseTypeDef1 Response 1.

kSDSPI_ResponseTypeDef1b Response 1 with busy.

kSDSPI_ResponseTypeDef2 Response 2.

kSDSPI_ResponseTypeDef3 Response 3.

kSDSPI_ResponseTypeDef7 Response 7.

41.3.4 enum _sdspi_cmd

Enumerator

kSDSPI_CmdGoIdle command go idle

kSDSPI_CmdCrc command crc protection

kSDSPI_CmdSendInterfaceCondition command send interface condition

41.4 Function Documentation

41.4.1 status_t SDSPI_Init (*sdspi_card_t* * *card*)

This function initializes the card on a specific SPI instance.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

Return values

Function Documentation

<i>kStatus_SDSPI_SetFrequencyFailed</i>	Set frequency failed.
<i>kStatus_SDSPI_GoIdleFailed</i>	Go idle failed.
<i>kStatus_SDSPI_SendInterfaceConditionFailed</i>	Send interface condition failed.
<i>kStatus_SDSPI_SendOperationConditionFailed</i>	Send operation condition failed.
<i>kStatus_Timeout</i>	Send command timeout.
<i>kStatus_SDSPI_NotSupportYet</i>	Not support yet.
<i>kStatus_SDSPI_ReadOcrFailed</i>	Read OCR failed.
<i>kStatus_SDSPI_SetBlockSizeFailed</i>	Set block size failed.
<i>kStatus_SDSPI_SendCsdFailed</i>	Send CSD failed.
<i>kStatus_SDSPI_SendCidFailed</i>	Send CID failed.
<i>kStatus_Success</i>	Operate successfully.

41.4.2 void SDSPI_Deinit (*sdspi_card_t * card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

41.4.3 bool SDSPI_CheckReadOnly (*sdspi_card_t * card*)

This function checks if the card is write-protected via CSD register.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

41.4.4 status_t SDSPI_ReadBlocks (*sdspi_card_t * card, uint8_t * buffer, uint32_t startBlock, uint32_t blockCount*)

This function reads blocks from specific card.

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	the buffer to hold the data read from card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to read

Return values

<i>kStatus_SDSPI_Send-CommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_Read-Failed</i>	Read data failed.
<i>kStatus_SDSPI_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

41.4.5 status_t SDSPI_WriteBlocks (*sdspi_card_t * card, uint8_t * buffer, uint32_t startBlock, uint32_t blockCount*)

This function writes blocks to specific card

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	the buffer holding the data to be written to the card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to write

Return values

<i>kStatus_SDSPI_WriteProtected</i>	Card is write protected.
<i>kStatus_SDSPI_SendCommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_ResponseError</i>	Response is error.
<i>kStatus_SDSPI_WriteFailed</i>	Write data failed.
<i>kStatus_SDSPI_ExchangeFailed</i>	Exchange data over SPI failed.
<i>kStatus_SDSPI_WaitReadyFailed</i>	Wait card to be ready status failed.
<i>kStatus_Success</i>	Operate successfully.

41.4.6 **status_t SDSPI_SendCid(sdspi_card_t * *card*)**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDSPI_SendCommandFailed</i>	Send command failed.
--	----------------------

<i>kStatus_SDSPI_ReadFailed</i>	Read data blocks failed.
<i>kStatus_Success</i>	Operate successfully.

41.4.7 **status_t SDSPI_SendPreErase (*sdspi_card_t * card, uint32_t blockCount*)**

This function should be called before SDSPI_WriteBlocks, it is used to set the number of the write blocks to be pre-erased before writing.

Parameters

<i>card</i>	Card descriptor.
<i>blockCount</i>	the block counts to be write.

Return values

<i>kStatus_SDSPI_SendCommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_SendApplicationCommandFailed</i>	
<i>kStatus_SDSPI_ResponseError</i>	
<i>kStatus_Success</i>	Operate successfully.

41.4.8 **status_t SDSPI_EraseBlocks (*sdspi_card_t * card, uint32_t startBlock, uint32_t blockCount*)**

Parameters

<i>card</i>	Card descriptor.
<i>startBlock</i>	start block address to be erase.
<i>blockCount</i>	the block counts to be erase.

Function Documentation

Return values

<i>kStatus_SDSPI_WaitReadyFailed</i>	Wait ready failed.
<i>kStatus_SDSPI_SendCommandFailed</i>	Send command failed.
<i>kStatus_Success</i>	Operate successfully.

41.4.9 **status_t SDSPI_SwitchToHighSpeed (*sdspi_card_t * card*)**

This function can be called after SDSPI_Init function if target board's layout support >25MHZ spi bau-drate, otherwise this function is useless. Be careful with call this function, code size and stack usage will be enlarge.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_Fail</i>	switch failed.
<i>kStatus_Success</i>	Operate successfully.

Chapter 42

Debug Console

42.1 Overview

This chapter describes the programming interface of the debug console driver.

The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data.

42.2 Function groups

42.2.1 Initialization

To initialize the debug console, call the [DbgConsole_Init\(\)](#) function with these parameters. This function automatically enables the module and the clock.

```
status_t DbgConsole_Init(uint8_t instance, uint32_t baudRate, serial_port_type_t device,  
                        uint32_t clkSrcFreq);
```

Selects the supported debug console hardware device type, such as

```
typedef enum _serial_port_type  
{  
    kSerialPort_Uart = 1U,  
    kSerialPort_UsbCdc,  
    kSerialPort_Swo,  
} serial_port_type_t;
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral.

This example shows how to call the [DbgConsole_Init\(\)](#) given the user configuration structure.

```
DbgConsole_Init(BOARD_DEBUG_UART_INSTANCE, BOARD_DEBUG_UART_BAUDRATE, BOARD_DEBUG_UART_TYPE,  
                BOARD_DEBUG_UART_CLK_FREQ);
```

42.2.2 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

Function groups

flags	Description
-	Left-justified within the given field width. Right-justified is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is written, a blank space is inserted before the value.
#	Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
Do not support	

specifier	Description
d or i	Signed decimal integer
f	Decimal floating point
F	Decimal floating point capital letters
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer capital letters
o	Signed octal
b	Binary value
p	Pointer address
u	Unsigned decimal integer
c	Character
s	String of characters
n	Nothing printed

Function groups

- Support a format specifier for SCANF following this prototype " %[*][width][length]specifier", which is explained below

*	Description
An optional starting asterisk indicates that the data is to be read from the stream but ignored. In other words, it is not stored in the corresponding argument.	

width	Description
This specifies the maximum number of characters to be read in the current reading operation.	

length	Description
hh	The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).
h	The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).
l	The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
ll	The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).
j or z or t	Not supported

specifier	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *

specifier	Qualifying Input	Type of argument
i	Integer: : Number optionally preceded with a + or - sign	int *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
a, A, e, E, f, F, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).	char *
u	Unsigned decimal integer.	unsigned int *

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file.

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE      /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF            DbgConsole_Printf
#define SCANF              DbgConsole_Scanf
#define PUTCHAR            DbgConsole_Putchar
#define GETCHAR            DbgConsole_Getchar
#else                      /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF            printf
#define SCANF              scanf
#define PUTCHAR            putchar
#define GETCHAR            getchar
#endif /* SDK_DEBUGCONSOLE */
```

42.3 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 MCUXpresso SDK __assert_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \"% %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file
           , line, func);
    for (;;) {}
}
```

Note:

To use 'printf' and 'scanf' for GNUC Base, add file '**fsl_sbrk.c**' in path: ..\{package}\devices\{subset}\utilities\fsl-sbrk.c to your project.

Modules

- [SWO](#)
- /*!
- [Semihosting](#)

Macros

- `#define DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN 0U`
Definition select redirect toolchain printf, scanf to uart or not.
- `#define DEBUGCONSOLE_REDIRECT_TO_SDK 1U`
Select SDK version printf, scanf.
- `#define DEBUGCONSOLE_DISABLE 2U`
Disable debugconsole function.
- `#define SDK_DEBUGCONSOLE 1U`
Definition to select sdk or toolchain printf, scanf.
- `#define SDK_DEBUGCONSOLE_UART`
Definition to select redirect toolchain printf, scanf to uart or not.
- `#define PRINTF DbgConsole_Printf`
Definition to select redirect toolchain printf, scanf to uart or not.

Typedefs

- `typedef void(* printfCb)(char *buf, int32_t *indicator, char val, int len)`
A function pointer which is used when format printf log.

Functions

- `int StrFormatPrintf (const char *fmt, va_list ap, char *buf, printfCb cb)`
This function outputs its parameters according to a formatted string.
- `int StrFormatScanf (const char *line_ptr, char *format, va_list args_ptr)`
Converts an input line of ASCII characters based upon a provided string format.

Initialization

- `status_t DbgConsole_Init (uint8_t instance, uint32_t baudRate, serial_port_type_t device, uint32_t clkSrcFreq)`
Initializes the peripheral used for debug messages.
- `status_t DbgConsole_Deinit (void)`
De-initializes the peripheral used for debug messages.
- `int DbgConsole_Printf (const char *formatString,...)`
Writes formatted output to the standard output stream.
- `int DbgConsole_Putchar (int ch)`
Writes a character to stdout.
- `int DbgConsole_Scanf (char *formatString,...)`
Reads formatted data from the standard input stream.
- `int DbgConsole_Getchar (void)`
Reads a character from standard input.
- `status_t DbgConsole_Flush (void)`
Debug console flush.

42.4 Macro Definition Documentation

42.4.1 #define DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN 0U

Select toolchain printf and scanf.

Function Documentation

42.4.2 #define DEBUGCONSOLE_REDIRECT_TO_SDK 1U

42.4.3 #define DEBUGCONSOLE_DISABLE 2U

42.4.4 #define SDK_DEBUGCONSOLE 1U

42.4.5 #define SDK_DEBUGCONSOLE_UART

42.4.6 #define PRINTF DbgConsole_Printf

if SDK_DEBUGCONSOLE defined to 0,it represents select toolchain printf, scanf. if SDK_DEBUGCONSOLE defined to 1,it represents select SDK version printf, scanf. if SDK_DEBUGCONSOLE defined to 2,it represents disable debugconsole function.

42.5 Function Documentation

42.5.1 status_t DbgConsole_Init (uint8_t *instance*, uint32_t *baudRate*, serial_port_type_t *device*, uint32_t *clkSrcFreq*)

Call this function to enable debug log messages to be output via the specified peripheral initialized by the serial manager module. After this function has returned, stdout and stdin are connected to the selected peripheral.

Parameters

<i>instance</i>	The instance of the module.
<i>baudRate</i>	The desired baud rate in bits per second.
<i>device</i>	Low level device type for the debug console, can be one of the following. <ul style="list-style-type: none">• kSerialPort_Uart,• kSerialPort_UsbCdc.
<i>clkSrcFreq</i>	Frequency of peripheral source clock.

Returns

Indicates whether initialization was successful or not.

Return values

<i>kStatus_Success</i>	Execution successfully
------------------------	------------------------

42.5.2 **status_t DbgConsole_Deinit (void)**

Call this function to disable debug log messages to be output via the specified peripheral initialized by the serial manager module.

Returns

Indicates whether de-initialization was successful or not.

42.5.3 **int DbgConsole_Printf (const char * *formatString*, ...)**

Call this function to write a formatted output to the standard output stream.

Parameters

<i>formatString</i>	Format control string.
---------------------	------------------------

Returns

Returns the number of characters printed or a negative value if an error occurs.

42.5.4 **int DbgConsole_Putchar (int *ch*)**

Call this function to write a character to stdout.

Parameters

<i>ch</i>	Character to be written.
-----------	--------------------------

Returns

Returns the character written.

Function Documentation

42.5.5 int DbgConsole_Scanf (*char *formatString*, ...)

Call this function to read formatted data from the standard input stream.

Note

Due the limitation in the BM OSA environment (CPU is blocked in the function, other tasks will not be scheduled), the function cannot be used when the DEBUG_CONSOLE_TRANSFER_NON_B-LOCKING is set in the BM OSA environment. And an error is returned when the function called in this case. The suggestion is that polling the non-blocking function DbgConsole_TryGetchar to get the input char.

Parameters

<i>formatString</i>	Format control string.
---------------------	------------------------

Returns

Returns the number of fields successfully converted and assigned.

42.5.6 int DbgConsole_Getchar (void)

Call this function to read a character from standard input.

Note

Due the limitation in the BM OSA environment (CPU is blocked in the function, other tasks will not be scheduled), the function cannot be used when the DEBUG_CONSOLE_TRANSFER_NON_B-LOCKING is set in the BM OSA environment. And an error is returned when the function called in this case. The suggestion is that polling the non-blocking function DbgConsole_TryGetchar to get the input char.

Returns

Returns the character read.

42.5.7 status_t DbgConsole_Flush (void)

Call this function to wait the tx buffer empty. If interrupt transfer is using, make sure the global IRQ is enable before call this function This function should be called when 1, before enter power down mode 2, log is required to print to terminal immediately

Returns

Indicates whether wait idle was successful or not.

42.5.8 int StrFormatPrintf (const char * *fmt*, va_list *ap*, char * *buf*, printfCb *cb*)

Note

I/O is performed by calling given function pointer using following (*func_ptr)(c);

Parameters

in	<i>fmt</i>	Format string for printf.
in	<i>ap</i>	Arguments to printf.
in	<i>buf</i>	pointer to the buffer
	<i>cb</i>	print callbk function pointer

Returns

Number of characters to be print

42.5.9 int StrFormatScanf (const char * *line_ptr*, char * *format*, va_list *args_ptr*)

Parameters

in	<i>line_ptr</i>	The input line of ASCII data.
in	<i>format</i>	Format first points to the format string.
in	<i>args_ptr</i>	The list of parameters.

Returns

Number of input items converted and assigned.

Return values

<i>IO_EOF</i>	When line_ptr is empty string "".
---------------	-----------------------------------

42.6 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism can be used, for example, to enable functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

42.6.1 Guide Semihosting for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging, if you want use PRINTF with semihosting, please make sure the `SDK_DEBUGCONSOLE` is disabled.

Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

Step 3: Starting semihosting

1. Choose "Semihosting_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Choose tab "General Options" -> "Library Configurations", select Semihosted, select Via semihosting.
1. Make sure the `SDK_DEBUGCONSOLE_UART` is not defined, remove the default definition in `fsl_debug_console.h`.
1. Start the project by choosing Project>Download and Debug.
2. Choose View>Terminal I/O to display the output from the I/O operations.

42.6.2 Guide Semihosting for Keil µVision

NOTE: Semihosting is not support by MDK-ARM, use the retargeting functionality of MDK-ARM instead.

42.6.3 Guide Semihosting for MCUXpresso IDE

Step 1: Setting up the environment

1. To set debugger options, choose Project>Properties. select the setting category.
2. Select Tool Settings, unfold MCU C Compile.
3. Select Preprocessor item.
4. Set SDK_DEBUGCONSOLE=0, if set SDK_DEBUGCONSOLE=1, the log will be redirect to the UART.

Step 2: Building the project

1. Compile and link the project.

Step 3: Starting semihosting

1. Download and debug the project.
2. When the project runs successfully, the result can be seen in the Console window.

Semihosting can also be selected through the "Quick settings" menu in the left bottom window, Quick settings->SDK Debug Console->Semihost console.

42.6.4 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
 - "Host Name (or IP address)" : localhost
 - "Port" :2333
 - "Connection type" : Telet.
 - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__heap_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__heap_size__=0x2000")
```

Step 2: Building the project

1. Change "CMakeLists.txt":

```
Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=nano.specs")"
to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=rdimon.specs")"
```

Replace paragraph

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mapcs")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} muldefs")
```

To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --specs=rdimon.specs ")
```

Remove

```
target_link_libraries(semihosting_ARMGCC.elf debug nosys)
```

2. Run "build_debug.bat" to build project

Step 3: Starting semihosting

- (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 = MK64FN1M0xxxx12
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.

42.7 SWO

/*!

Serial wire output is a mechanism for ARM targets to output signal from core through a single pin. Some IDE support SWO also, such IAR and KEIL, both input and output are supported, reference below for detail.

42.7.1 Guide SWO for SDK

NOTE: After the setting both "printf" and "PRINTF" are available for debugging, JlinkSWOViewer can be used to capture the output log.

Step 1: Setting up the environment

1. Define DEBUG_CONSOLE_IO_SWO in your project settings.
2. Prepare code, the port and baudrate can be decided by application, clkSrcFreq should be mcu core clock frequency:

```
DbgConsole_Init(port, baudrate, DEBUG_CONSOLE_DEVICE_TYPE_SWO, clkSrcFreq);
```

3. Use PRINTF or printf to print some thing in application.

Step 2: Building the project

Step 3: Download and run project

42.7.1.1 Guide SWO for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. Choose project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Choose tab "General Options" -> "Library Configurations", select Semihosted, select Via SWO.
4. To configure the hardware's generation of trace data, click the SWO Configuration button available in the SWO Configuration dialog box. The value of the CPU clock option must reflect the frequency of the CPU clock speed at which the application executes. Note also that the settings you make are preserved between debug sessions. To decrease the amount of transmissions on the communication channel, you can disable the Timestamp option. Alternatively, set a lower rate for PC Sampling or use a higher SWO clock frequency.
5. Open the SWO Trace window from J-LINK, and click the Activate button to enable trace data collection.

6. There are three cases for this SDK_DEBUGCONSOLE_UART whether or not defined. a: if use uppercase PRINTF to output log,The SDK_DEBUGCONSOLE_UART defined or not defined will not effect debug function. b: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to zero,then debug function ok. c: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to one,then debug function ok.

NOTE: Case a or c only apply at example which enable swo function,the SDK_DEBUGCONSOLE_UART definition in fsl_debug_console.h. For case a and c, Do and not do the above third step will be not affect function.

1. Start the project by choosing Project>Download and Debug.

Step 2: Building the project

Step 3: Starting swo

1. Download and debug application.
2. Choose View -> Terminal I/O to display the output from the I/O operations.
3. Run application.

42.7.2 Guide SWO for Keil µVision

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. There are three cases for this SDK_DEBUGCONSOLE_UART whether or not defined. a: if use uppercase PRINTF to output log,the SDK_DEBUGCONSOLE_UART definition does not affect the functionality and skip the second step directly. b: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to zero,then start the second step. c: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to one,then skip the second step directly.

NOTE: Case a or c only apply at example which enable swo function,the SDK_DEBUGCONSOLE_UART definition in fsl_debug_console.h.

1. In menu bar, click Management Run-Time Environment icon, select Compiler, unfold I/O, enable STDERR/STDIN/STDOUT and set the variant to ITM.
2. Open Project>Options for target or using Alt+F7 or click.
3. Select “Debug” tab, select “J-Link/J-Trace Cortex” and click “Setting button”.
4. Select “Debug” tab and choose Port:SW, then select “Trace” tab, choose “Enable” and click OK, please make sure the Core clock is set correctly, enable autodetect max SWO clk, enable ITM Stimulus Ports 0.

Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

SWO

Step 4: Run the project

1. Choose "Debug" on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

42.7.3 Guide SWO for MCUXpresso IDE

NOTE: MCUX support SWO for LPC-Link2 debug probe only.

42.7.4 Guide SWO for ARMGCC

NOTE: ARMGCC has no library support SWO.

Chapter 43

Notification Framework

43.1 Overview

This section describes the programming interface of the Notifier driver.

43.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

These are the steps for the configuration transition.

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.
The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending a "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.
2. After the "BEFORE" message is processed successfully, the system switches to the new configuration.
3. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This example shows how to use the Notifier in the Power Manager application.

```
#include "fsl_notifier.h"

// Definition of the Power Manager callback.
status_t callback0(notifier_notification_block_t *notify, void *data)
{
    status_t ret = kStatus_Success;

    ...
    ...

    return ret;
}
// Definition of the Power Manager user function.
status_t APP_PowerModeSwitch(notifier_user_config_t *targetConfig, void *userData)
```

Notifier Overview

```
...
...
...
}
...
...
...
...
...
...
...
// Main function.
int main(void)
{
    // Define a notifier handle.
    notifier_handle_t powerModeHandle;

    // Callback configuration.
    user_callback_data_t callbackData0;

    notifier_callback_config_t callbackCfg0 = {callback0,
                                              kNOTIFIER_CallbackBeforeAfter,
                                              (void *)&callbackData0};

    notifier_callback_config_t callbacks[] = {callbackCfg0};

    // Power mode configurations.
    power_user_config_t vlprConfig;
    power_user_config_t stopConfig;

    notifier_user_config_t *powerConfigs[] = {&vlprConfig, &stopConfig};

    // Definition of a transition to and out the power modes.
    vlprConfig.mode = kAPP_PowerModeVlpr;
    vlprConfig.enableLowPowerWakeUpOnInterrupt = false;

    stopConfig = vlprConfig;
    stopConfig.mode = kAPP_PowerModeStop;

    // Create Notifier handle.
    NOTIFIER_CreateHandle(&powerModeHandle, powerConfigs, 2U, callbacks, 1U,
                          APP_PowerModeSwitch, NULL);
    ...
    ...
    // Power mode switch.
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex,
                          kNOTIFIER_PolicyAgreement);
}
```

Data Structures

- struct **notifier_notification_block_t**
notification block passed to the registered callback function. [More...](#)
- struct **notifier_callback_config_t**
Callback configuration structure. [More...](#)
- struct **notifier_handle_t**
Notifier handle structure. [More...](#)

Typedefs

- **typedef void notifier_user_config_t**
Notifier user configuration type.
- **typedef status_t(* notifier_user_function_t)(notifier_user_config_t *targetConfig, void *userData)**
Notifier user function prototype Use this function to execute specific operations in configuration switch.

- `typedef status_t(* notifier_callback_t)(notifier_notification_block_t *notify, void *data)`
Callback prototype.

Enumerations

- `enum _notifier_status {`
 `kStatus_NOTIFIER_ErrorNotificationBefore,`
 `kStatus_NOTIFIER_ErrorNotificationAfter }`
Notifier error codes.
- `enum notifier_policy_t {`
 `kNOTIFIER_PolicyAgreement,`
 `kNOTIFIER_PolicyForcible }`
Notifier policies.
- `enum notifier_notification_type_t {`
 `kNOTIFIER_NotifyRecover = 0x00U,`
 `kNOTIFIER_NotifyBefore = 0x01U,`
 `kNOTIFIER_NotifyAfter = 0x02U }`
Notification type.
- `enum notifier_callback_type_t {`
 `kNOTIFIER_CallbackBefore = 0x01U,`
 `kNOTIFIER_CallbackAfter = 0x02U,`
 `kNOTIFIER_CallbackBeforeAfter = 0x03U }`
The callback type, which indicates kinds of notification the callback handles.

Functions

- `status_t NOTIFIER_CreateHandle (notifier_handle_t *notifierHandle, notifier_user_config_t **configs, uint8_t configsNumber, notifier_callback_config_t *callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void *userData)`
Creates a Notifier handle.
- `status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)`
Switches the configuration according to a pre-defined structure.
- `uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)`
This function returns the last failed notification callback.

43.3 Data Structure Documentation

43.3.1 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

43.3.1.0.0.81 Field Documentation

43.3.1.0.0.81.1 `notifier_user_config_t* notifier_notification_block_t::targetConfig`

43.3.1.0.0.81.2 `notifier_policy_t notifier_notification_block_t::policy`

43.3.1.0.0.81.3 `notifier_notification_type_t notifier_notification_block_t::notifyType`

43.3.2 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.

43.3.2.0.0.82 Field Documentation

43.3.2.0.0.82.1 `notifier_callback_t notifier_callback_config_t::callback`

43.3.2.0.0.82.2 `notifier_callback_type_t notifier_callback_config_t::callbackType`

43.3.2.0.0.82.3 `void* notifier_callback_config_t::callbackData`

43.3.3 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.

43.3.3.0.0.83 Field Documentation

43.3.3.0.0.83.1 `notifier_user_config_t** notifier_handle_t::configsTable`

43.3.3.0.0.83.2 `uint8_t notifier_handle_t::configsNumber`

43.3.3.0.0.83.3 `notifier_callback_config_t* notifier_handle_t::callbacksTable`

43.3.3.0.0.83.4 `uint8_t notifier_handle_t::callbacksNumber`

43.3.3.0.0.83.5 `uint8_t notifier_handle_t::errorCallbackIndex`

43.3.3.0.0.83.6 `uint8_t notifier_handle_t::currentConfigIndex`

43.3.3.0.0.83.7 `notifier_user_function_t notifier_handle_t::userFunction`

43.3.3.0.0.83.8 `void* notifier_handle_t::userData`

43.4 Typedef Documentation

43.4.1 `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.

43.4.2 `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.

43.4.3 **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.

43.5 Enumeration Type Documentation

43.5.1 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.

43.5.2 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.

43.5.3 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.

43.5.4 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

43.6 Function Documentation

43.6.1 `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.

43.6.2 **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.

43.6.3 **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 44

Shell

44.1 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.

44.2 Function groups

44.2.1 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>> ");
```

44.2.2 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.

44.2.3 Shell Operation

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
SHELL_Main(&user_context);
```

Data Structures

- struct [shell_command_t](#)
User command data configuration structure. [More...](#)

Macros

- #define [SHELL_NON_BLOCKING_MODE](#) SERIAL_MANAGER_NON_BLOCKING_MODE
Whether use non-blocking mode.
- #define [SHELL_AUTO_COMPLETE](#) (1U)
Macro to set on/off auto-complete feature.
- #define [SHELL_BUFFER_SIZE](#) (64U)
Macro to set console buffer size.
- #define [SHELL_MAX_ARGS](#) (8U)
Macro to set maximum arguments in command.
- #define [SHELL_HISTORY_COUNT](#) (3U)
Macro to set maximum count of history commands.
- #define [SHELL_IGNORE_PARAMETER_COUNT](#) (0xFF)
Macro to bypass arguments check.
- #define [SHELL_HANDLE_SIZE](#) (520U)
The handle size of the shell module.
- #define [SHELL_COMMAND_DEFINE](#)(command, descriptor, callback, paramInt)
Defines the shell command structure.
- #define [SHELL_COMMAND](#)(command) &g_shellCommand##command
Gets the shell command pointer.

Typedefs

- typedef void * [shell_handle_t](#)
The handle of the shell module.
- typedef [shell_status_t](#)(* [cmd_function_t](#))(shell_handle_t shellHandle, int32_t argc, char **argv)
User command function prototype.

Enumerations

- enum `shell_status_t` {

 `kStatus_SHELL_Success` = `kStatus_Success`,

 `kStatus_SHELL_Error` = `MAKE_STATUS(kStatusGroup_SHELL, 1)`,

 `kStatus_SHELL_OpenWriteHandleFailed` = `MAKE_STATUS(kStatusGroup_SHELL, 2)`,

 `kStatus_SHELL_OpenReadHandleFailed` = `MAKE_STATUS(kStatusGroup_SHELL, 3)` }

Shell functional operation

- `shell_status_t SHELL_Init (shell_handle_t shellHandle, serial_handle_t serialHandle, char *prompt)`

Initializes the shell module.
- `shell_status_t SHELL_RegisterCommand (shell_handle_t shellHandle, shell_command_t *command)`

Registers the shell command.
- `shell_status_t SHELL_UnregisterCommand (shell_command_t *command)`

Unregisters the shell command.
- `shell_status_t SHELL_Write (shell_handle_t shellHandle, char *buffer, uint32_t length)`

Sends data to the shell output stream.
- `int SHELL_Printf (shell_handle_t shellHandle, const char *formatString,...)`

Writes formatted output to the shell output stream.
- `void SHELL_Task (shell_handle_t shellHandle)`

The task function for Shell.

44.3 Data Structure Documentation

44.3.1 struct shell_command_t

Data Fields

- `const char * pcCommand`

The command that is executed.
- `char * pcHelpString`

String that describes how to use the command.
- `const cmd_function_t pFuncCallBack`

A pointer to the callback function that returns the output generated by the command.
- `uint8_t cExpectedNumberOfParameters`

Commands expect a fixed number of parameters, which may be zero.
- `list_element_t link`

link of the element

44.3.1.0.0.84 Field Documentation

44.3.1.0.0.84.1 const char* shell_command_t::pcCommand

For example "help". It must be all lower case.

Macro Definition Documentation

44.3.1.0.0.84.2 `char* shell_command_t::pcHelpString`

It should start with the command itself, and end with "\r\n". For example "help: Returns a list of all the commands\r\n".

44.3.1.0.0.84.3 `const cmd_function_t shell_command_t::pFuncCallBack`

44.3.1.0.0.84.4 `uint8_t shell_command_t::cExpectedNumberOfParameters`

44.4 Macro Definition Documentation

44.4.1 `#define SHELL_NON_BLOCKING_MODE SERIAL_MANAGER_NON_BLOCKING_MODE`

44.4.2 `#define SHELL_AUTO_COMPLETE (1U)`

44.4.3 `#define SHELL_BUFFER_SIZE (64U)`

44.4.4 `#define SHELL_MAX_ARGS (8U)`

44.4.5 `#define SHELL_HISTORY_COUNT (3U)`

44.4.6 `#define SHELL_HANDLE_SIZE (520U)`

It is the sum of the SHELL_HISTORY_COUNT * SHELL_BUFFER_SIZE + SHELL_BUFFER_SIZE + SERIAL_MANAGER_READ_HANDLE_SIZE + SERIAL_MANAGER_WRITE_HANDLE_SIZE

44.4.7 `#define SHELL_COMMAND_DEFINE(command, descriptor, callback, paramInt)`

Value:

```
\nshell_command_t g_shellCommand##command = {\n    (#command), (descriptor), (callback), (paramCount), {0},\n}
```

This macro is used to define the shell command structure `shell_command_t`. And then uses the macro SHELL_COMMAND to get the command structure pointer. The macro should not be used in any function.

This is a example,

```
* SHELL_COMMAND_DEFINE(exit, "\r\n\"exit\": Exit program\r\n", SHELL_ExitCommand, 0);\n* SHELL_RegisterCommand(s_shellHandle, SHELL_COMMAND(exit));\n*
```

Parameters

<i>command</i>	The command string of the command. The double quotes do not need. Such as exit for "exit", help for "Help", read for "read".
<i>descriptor</i>	The description of the command is used for showing the command usage when "help" is typing.
<i>callback</i>	The callback of the command is used to handle the command line when the input command is matched.
<i>paramCount</i>	The max parameter count of the current command.

44.4.8 #define SHELL_COMMAND(*command*) &g_shellCommand##*command*

This macro is used to get the shell command pointer. The macro should not be used before the macro SHELL_COMMAND_DEFINE is used.

Parameters

<i>command</i>	The command string of the command. The double quotes do not need. Such as exit for "exit", help for "Help", read for "read".
----------------	--

44.5 Typedef Documentation

44.5.1 **typedef shell_status_t(* cmd_function_t)(shell_handle_t shellHandle, int32_t argc, char **argv)**

44.6 Enumeration Type Documentation

44.6.1 enum shell_status_t

Enumerator

kStatus_SHELL_Success Success.

kStatus_SHELL_Error Failed.

kStatus_SHELL_OpenWriteHandleFailed Open write handle failed.

kStatus_SHELL_OpenReadHandleFailed Open read handle failed.

44.7 Function Documentation

44.7.1 **shell_status_t SHELL_Init (shell_handle_t shellHandle, serial_handle_t serialHandle, char * prompt)**

This function must be called before calling all other Shell functions. Call operation the Shell commands with user-defined settings. The example below shows how to set up the Shell and how to call the SHELL_Init function by passing in these parameters. This is an example.

Function Documentation

```
* static uint8_t s_shellHandleBuffer[SHELL_HANDLE_SIZE];
* static shell_handle_t s_shellHandle = &s_shellHandleBuffer[0];
* SHELL_Init(s_shellHandle, s_serialHandle, "Test@SHELL>");
*
```

Parameters

<i>shellHandle</i>	Pointer to point to a memory space of size SHELL_HANDLE_SIZE allocated by the caller.
<i>serialHandle</i>	The serial manager module handle pointer.
<i>prompt</i>	The string prompt pointer of Shell. Only the global variable can be passed.

Return values

<i>kStatus_SHELL_Success</i>	The shell initialization succeed.
<i>kStatus_SHELL_Error</i>	An error occurred when the shell is initialized.
<i>kStatus_SHELL_Open-WriteHandleFailed</i>	Open the write handle failed.
<i>kStatus_SHELL_Open-ReadHandleFailed</i>	Open the read handle failed.

44.7.2 **shell_status_t SHELL_RegisterCommand (shell_handle_t *shellHandle*, shell_command_t * *command*)**

This function is used to register the shell command by using the command configuration #shell_command_config_t. This is a example,

```
* SHELL_COMMAND_DEFINE(exit, "\r\n\"exit\": Exit program\r\n", SHELL_ExitCommand, 0);
* SHELL_RegisterCommand(s_shellHandle, SHELL_COMMAND(exit));
*
```

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>command</i>	The command element.

Return values

<i>kStatus_SHELL_Success</i>	Successfully register the command.
<i>kStatus_SHELL_Error</i>	An error occurred.

44.7.3 shell_status_t SHELL_UnregisterCommand (shell_command_t * *command*)

This function is used to unregister the shell command.

Parameters

<i>command</i>	The command element.
----------------	----------------------

Return values

<i>kStatus_SHELL_Success</i>	Successfully unregister the command.
------------------------------	--------------------------------------

44.7.4 shell_status_t SHELL_Write (shell_handle_t *shellHandle*, char * *buffer*, uint32_t *length*)

This function is used to send data to the shell output stream.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>buffer</i>	Start address of the data to write.
<i>length</i>	Length of the data to write.

Return values

<i>kStatus_SHELL_Success</i>	Successfully send data.
<i>kStatus_SHELL_Error</i>	An error occurred.

44.7.5 int SHELL_Printf (shell_handle_t *shellHandle*, const char * *formatString*, ...)

Call this function to write a formatted output to the shell output stream.

Function Documentation

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>formatString</i>	Format string.

Returns

Returns the number of characters printed or a negative value if an error occurs.

44.7.6 void SHELL_Task (**shell_handle_t shellHandle**)

The task function for Shell; The function should be polled by upper layer. This function does not return until Shell command exit was called.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
--------------------	----------------------------------

Chapter 45

Ftfx_cache_driver

45.1 Overview

Data Structures

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

Enumerations

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

Functions

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

FTFx cache version

- #define `FSL_FTFX_CACHE_DRIVER_VERSION` (MAKE_VERSION(1, 0, 0))
Flexnvm driver version for SDK.

45.2 Data Structure Documentation

45.2.1 struct `ftfx_prefetch_speculation_status_t`

Data Fields

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

Function Documentation

45.2.1.0.0.85 Field Documentation

45.2.1.0.0.85.1 `bool ftfx_prefetch_speculation_status_t::instructionOff`

45.2.1.0.0.85.2 `bool ftfx_prefetch_speculation_status_t::dataOff`

45.2.2 `struct ftfx_cache_config_t`

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

Data Fields

- `uint8_t flashMemoryIndex`
0 - primary flash; 1 - secondary flash
- `uint32_t * comBitOperFuncAddr`
An buffer point to the flash execute-in-RAM function.

45.2.2.0.0.86 Field Documentation

45.2.2.0.0.86.1 `uint32_t* ftfx_cache_config_t::comBitOperFuncAddr`

45.3 Macro Definition Documentation

45.3.1 `#define FSL_FTFX_CACHE_DRIVER_VERSION (MAKE_VERSION(1, 0, 0))`

Version 1.0.0.

45.4 Enumeration Type Documentation

45.4.1 `enum _ftfx_cache_ram_func_constants`

Enumerator

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

45.5 Function Documentation

45.5.1 `status_t FTFx_CACHE_Init (ftfx_cache_config_t * config)`

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

Parameters

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

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	An invalid argument is provided.
#kStatus_FTFx_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.

45.5.2 status_t FTFx_CACHE_ClearCachePrefetchSpeculation (ftfx_cache_config_t * *config*, bool *isPreProcess*)

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>process</i>	The possible option used to control flash cache/prefetch/speculation

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	Invalid argument is provided.
#kStatus_FTFx_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.

45.5.3 status_t FTFx_CACHE_PflashSetPrefetchSpeculation (ftfx_prefetch_speculation_status_t * *speculationStatus*)

Parameters

<i>speculation-Status</i>	The expected protect status to set to the PFlash protection register. Each bit is
---------------------------	---

Function Documentation

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_Invalid-SpeculationOption</code>	An invalid speculation option argument is provided.

45.5.4 `status_t FTFx_CACHE_PflashGetPrefetchSpeculation (ftfx_prefetch_speculation_status_t * speculationStatus)`

Parameters

<code>speculation-Status</code>	Speculation status returned by the PFlash IP.
---------------------------------	---

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
------------------------------------	--------------------------------

Chapter 46

Ftfx_flash_driver

46.1 Overview

Data Structures

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

Enumerations

- enum `flash_prot_state_t` {
 `kFLASH_ProtectionStateUnprotected`,
 `kFLASH_ProtectionStateProtected`,
 `kFLASH_ProtectionStateMixed` }
Enumeration for the three possible flash protection levels.
- enum `flash_xacc_state_t` {
 `kFLASH_AccessStateUnLimited`,
 `kFLASH_AccessStateExecuteOnly`,
 `kFLASH_AccessStateMixed` }
Enumeration for the three possible flash execute access levels.
- enum `flash_property_tag_t` {
 `kFLASH_PropertyPflash0SectorSize` = 0x00U,
 `kFLASH_PropertyPflash0TotalSize` = 0x01U,
 `kFLASH_PropertyPflash0BlockSize` = 0x02U,
 `kFLASH_PropertyPflash0BlockCount` = 0x03U,
 `kFLASH_PropertyPflash0BlockBaseAddr` = 0x04U,
 `kFLASH_PropertyPflash0FacSupport` = 0x05U,
 `kFLASH_PropertyPflash0AccessSegmentSize` = 0x06U,
 `kFLASH_PropertyPflash0AccessSegmentCount` = 0x07U,
 `kFLASH_PropertyPflash1SectorSize` = 0x10U,
 `kFLASH_PropertyPflash1TotalSize` = 0x11U,
 `kFLASH_PropertyPflash1BlockSize` = 0x12U,
 `kFLASH_PropertyPflash1BlockCount` = 0x13U,
 `kFLASH_PropertyPflash1BlockBaseAddr` = 0x14U,
 `kFLASH_PropertyPflash1FacSupport` = 0x15U,
 `kFLASH_PropertyPflash1AccessSegmentSize` = 0x16U,
 `kFLASH_PropertyPflash1AccessSegmentCount` = 0x17U,
 `kFLASH_PropertyFlexRamBlockBaseAddr` = 0x20U,
 `kFLASH_PropertyFlexRamTotalSize` = 0x21U }
Enumeration for various flash properties.

Overview

Flash version

- enum `_flash_driver_version_constants` {
 `kFLASH_DriverVersionName` = 'F',
 `kFLASH_DriverVersionMajor` = 3,
 `kFLASH_DriverVersionMinor` = 0,
 `kFLASH_DriverVersionBugfix` = 0 }
 Flash driver version for ROM.
- #define `FSL_FLASH_DRIVER_VERSION` (MAKE_VERSION(3, 0, 0))
 Flash driver version for SDK.

Initialization

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

Erasing

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

Programming

Erases the entire flexnvm, including protected sectors.

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_EraseKeyError</code>	API erase key is invalid.

<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.
<code>#kStatus_FTFx_PartitionStatusUpdateFailure</code>	Failed to update the partition status.

- `status_t FLASH_Program (flash_config_t *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)`
Programs flash with data at locations passed in through parameters.
- `status_t FLASH_ProgramSection (flash_config_t *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)`
Programs flash with data at locations passed in through parameters via the Program Section command.

Reading

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

Verification

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

Security

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

FlexRAM

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

Overview

Protection

Swaps the lower half flash with the higher half flash.

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>isSetEnable</i>	The possible option used to configure the Flash Swap function or check the flash Swap status.

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_-AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_Swap-IndicatorAddressError</code>	Swap indicator address is invalid.
<code>#kStatus_FTFx_Execute-InRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_Access-Error</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_-ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_-CommandFailure</code>	Run-time error during command execution.
<code>#kStatus_FTFx_Swap-SystemNotInUninitialized</code>	Swap system is not in an uninitialized state.

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

Properties

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

46.2 Data Structure Documentation

46.2.1 union pflash_prot_status_t

Data Fields

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

46.2.1.0.0.87 Field Documentation

46.2.1.0.0.87.1 `uint32_t pflash_prot_status_t::protl`

46.2.1.0.0.87.2 `uint32_t pflash_prot_status_t::proth`

46.2.1.0.0.87.3 `uint8_t pflash_prot_status_t::protsl`

46.2.1.0.0.87.4 `uint8_t pflash_prot_status_t::protsh`

46.2.2 struct flash_config_t

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

46.3 Macro Definition Documentation

46.3.1 `#define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(3, 0, 0))`

Version 3.0.0.

46.4 Enumeration Type Documentation

46.4.1 enum _flash_driver_version_constants

Enumerator

kFLASH_DriverVersionName Flash driver version name.

Enumeration Type Documentation

kFLASH_DriverVersionMajor Major flash driver version.
kFLASH_DriverVersionMinor Minor flash driver version.
kFLASH_DriverVersionBugfix Bugfix for flash driver version.

46.4.2 enum flash_prot_state_t

Enumerator

kFLASH_ProtectionStateUnprotected Flash region is not protected.
kFLASH_ProtectionStateProtected Flash region is protected.
kFLASH_ProtectionStateMixed Flash is mixed with protected and unprotected region.

46.4.3 enum flash_xacc_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.

46.4.4 enum flash_property_tag_t

Enumerator

kFLASH_PropertyPflash0SectorSize Pflash sector size property.
kFLASH_PropertyPflash0TotalSize Pflash total size property.
kFLASH_PropertyPflash0BlockSize Pflash block size property.
kFLASH_PropertyPflash0BlockCount Pflash block count property.
kFLASH_PropertyPflash0BlockBaseAddr Pflash block base address property.
kFLASH_PropertyPflash0FacSupport Pflash fac support property.
kFLASH_PropertyPflash0AccessSegmentSize Pflash access segment size property.
kFLASH_PropertyPflash0AccessSegmentCount Pflash access segment count property.
kFLASH_PropertyPflash1SectorSize Pflash sector size property.
kFLASH_PropertyPflash1TotalSize Pflash total size property.
kFLASH_PropertyPflash1BlockSize Pflash block size property.
kFLASH_PropertyPflash1BlockCount Pflash block count property.
kFLASH_PropertyPflash1BlockBaseAddr Pflash block base address property.
kFLASH_PropertyPflash1FacSupport Pflash fac support property.
kFLASH_PropertyPflash1AccessSegmentSize Pflash access segment size property.
kFLASH_PropertyPflash1AccessSegmentCount Pflash access segment count property.
kFLASH_PropertyFlexRamBlockBaseAddr FlexRam block base address property.
kFLASH_PropertyFlexRamTotalSize FlexRam total size property.

46.5 Function Documentation

46.5.1 **status_t FLASH_Init (flash_config_t * *config*)**

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

Function Documentation

Parameters

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

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_InvalidArgument	An invalid argument is provided.
#kStatus_FTFx_ExecuteInRamFunctionNotReady	Execute-in-RAM function is not available.
#kStatus_FTFx_PartitionStatusUpdateFailure	Failed to update the partition status.

46.5.2 **status_t FLASH_Erase (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, uint32_t *key*)**

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

<i>config</i>	The pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
<i>key</i>	The value used to validate all flash erase APIs.

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_InvalidArgument	An invalid argument is provided.
#kStatus_FTFx_AlignmentError	The parameter is not aligned with the specified baseline.

<code>#kStatus_FTFx_Address_Error</code>	The address is out of range.
<code>#kStatus_FTFx_EraseKeyError</code>	The API erase key is invalid.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

46.5.3 `status_t FLASH_EraseAll(flash_config_t * config, uint32_t key)`

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_EraseKeyError</code>	API erase key is invalid.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.

Function Documentation

<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.
<code>#kStatus_FTFx_PartitionStatusUpdateFailure</code>	Failed to update the partition status.

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

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with the specified baseline.
<code>#kStatus_FTFx_AddressError</code>	Address is out of range.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.

<code>#kStatus_FTFx_- ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_- CommandFailure</code>	Run-time error during the command execution.

46.5.5 `status_t FLASH_ProgramSection (flash_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)`

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_Invalid- Argument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_- AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_Address- Error</code>	Address is out of range.
<code>#kStatus_FTFx_Set- FlexramAsRamError</code>	Failed to set flexram as RAM.
<code>#kStatus_FTFx_Execute- InRamFunctionNotReady</code>	Execute-in-RAM function is not available.

Function Documentation

<code>#kStatus_FTFx_Error</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.
<code>#kStatus_FTFx_RecoverFlexramAsEepromError</code>	Failed to recover FlexRAM as EEPROM.

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

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with the specified baseline.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.

<code>#kStatus_FTFx_Access-Error</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

46.5.7 `status_t FLASH_VerifyErase (flash_config_t * config, uint32_t start, uint32_t lengthInBytes, ftfx_margin_value_t margin)`

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

Parameters

<code>config</code>	A pointer to the storage for the driver runtime state.
<code>start</code>	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.
<code>lengthInBytes</code>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<code>margin</code>	Read margin choice.

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_Invalid-Argument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_Address-Error</code>	Address is out of range.
<code>#kStatus_FTFx_Execute-InRamFunctionNotReady</code>	Execute-in-RAM function is not available.

Function Documentation

<code>#kStatus_FTFx_Error</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

46.5.8 `status_t FLASH_VerifyEraseAll (flash_config_t * config, ftfx_margin_value_t margin)`

This function checks whether the flash is erased to the specified read margin level.

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_Error</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

46.5.9 `status_t FLASH_VerifyProgram (flash_config_t * config, uint32_t start, uint32_t lengthInBytes, const uint8_t * expectedData, ftfx_margin_value_t margin, uint32_t * failedAddress, uint32_t * failedData)`

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

Parameters

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

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_InvalidArgument	An invalid argument is provided.
#kStatus_FTFx_AlignmentError	Parameter is not aligned with specified baseline.
#kStatus_FTFx_AddressError	Address is out of range.
#kStatus_FTFx_ExecuteInRamFunctionNotReady	Execute-in-RAM function is not available.
#kStatus_FTFx_AccessError	Invalid instruction codes and out-of bounds addresses.
#kStatus_FTFx_ProtectionViolation	The program/erase operation is requested to execute on protected areas.
#kStatus_FTFx_CommandFailure	Run-time error during the command execution.

46.5.10 **status_t FLASH_GetSecurityState (flash_config_t * config, ftfx_security_state_t * state)**

This function retrieves the current flash security status, including the security enabling state and the back-door key enabling state.

Function Documentation

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

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	An invalid argument is provided.

46.5.11 **status_t FLASH_SecurityBypass (flash_config_t * config, const uint8_t * backdoorKey)**

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>backdoorKey</i>	A pointer to the user buffer containing the backdoor key.

Return values

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

46.5.12 **status_t FLASH_SetFlexramFunction (flash_config_t * config, ftfx_flexram_func_opt_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

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

46.5.13 `status_t FLASH_IsProtected (flash_config_t * config, uint32_t start, uint32_t lengthInBytes, flash_prot_state_t * protection_state)`

This function retrieves the current flash protect status for a given flash area as determined by the start address and length.

Parameters

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

Function Documentation

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_AddressError</code>	The address is out of range.

46.5.14 `status_t FLASH_IsExecuteOnly (flash_config_t * config, uint32_t start, uint32_t lengthInBytes, flash_xacc_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

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	The parameter is not aligned to the specified baseline.
<code>#kStatus_FTFx_AddressError</code>	The address is out of range.

46.5.15 status_t FLASH_PflashSetProtection (flash_config_t * *config*, pflash_prot_status_t * *protectStatus*)

Parameters

<i>config</i>	A pointer to storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the PFlash protection register. Each bit is corresponding to protection of 1/32(64) of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	An invalid argument is provided.
#kStatus_FTFx_-CommandFailure	Run-time error during command execution.

46.5.16 status_t FLASH_PflashGetProtection (flash_config_t * *config*, pflash_prot_status_t * *protectStatus*)

Parameters

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

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	An invalid argument is provided.

Function Documentation

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

Parameters

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

Return values

<i>#kStatus_FTFx_Success</i>	API was executed successfully.
<i>#kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>#kStatus_FTFx_-UnknownProperty</i>	An unknown property tag.

Function Documentation

Chapter 47

Ftfx_flexnvm_driver

47.1 Overview

Data Structures

- struct `flexnvm_config_t`
Flexnvm driver state information. [More...](#)

Enumerations

- enum `flexnvm_property_tag_t`{
 kFLEXNVM_PropertyDflashSectorSize = 0x00U,
 kFLEXNVM_PropertyDflashTotalSize = 0x01U,
 kFLEXNVM_PropertyDflashBlockSize = 0x02U,
 kFLEXNVM_PropertyDflashBlockCount = 0x03U,
 kFLEXNVM_PropertyDflashBlockBaseAddr = 0x04U,
 kFLEXNVM_PropertyFlexRamBlockBaseAddr = 0x05U,
 kFLEXNVM_PropertyFlexRamTotalSize = 0x06U,
 kFLEXNVM_PropertyEepromTotalSize = 0x07U }

Enumeration for various flexnvm properties.

Functions

- status_t `FLEXNVM_EepromWrite` (`flexnvm_config_t` *config, `uint32_t` start, `uint8_t` *src, `uint32_t` lengthInBytes)
Programs the EEPROM with data at locations passed in through parameters.

Flexnvm version

- #define `FSL_FLEXNVM_DRIVER_VERSION` (MAKE_VERSION(1, 0, 0))
Flexnvm driver version for SDK.

Initialization

- status_t `FLEXNVM_Init` (`flexnvm_config_t` *config)
Initializes the global flash properties structure members.

Erasing

- status_t `FLEXNVM_DflashErase` (`flexnvm_config_t` *config, `uint32_t` start, `uint32_t` lengthInBytes, `uint32_t` key)
Erases the Dflash sectors encompassed by parameters passed into function.
- status_t `FLEXNVM_EraseAll` (`flexnvm_config_t` *config, `uint32_t` key)
Erases entire flexnvm.

Overview

Programming

Erases the entire flexnvm, including protected sectors.

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

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_InvalidArgument	An invalid argument is provided.
#kStatus_FTFx_EraseKeyError	API erase key is invalid.
#kStatus_FTFx_ExecuteInRamFunctionNotReady	Execute-in-RAM function is not available.
#kStatus_FTFx_AccessError	Invalid instruction codes and out-of bounds addresses.
#kStatus_FTFx_ProtectionViolation	The program/erase operation is requested to execute on protected areas.
#kStatus_FTFx_CommandFailure	Run-time error during command execution.
#kStatus_FTFx_PartitionStatusUpdateFailure	Failed to update the partition status.

- status_t **FLEXNVM_DflashProgram** (*flexnvm_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)
Programs flash with data at locations passed in through parameters.
- status_t **FLEXNVM_DflashProgramSection** (*flexnvm_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)
Programs flash with data at locations passed in through parameters via the Program Section command.
- status_t **FLEXNVM_ProgramPartition** (*flexnvm_config_t* *config, ftfx_partition_flexram_load_opt_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)
Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Reading

- status_t **FLEXNVM_ReadResource** (*flexnvm_config_t* *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, ftfx_read_resource_opt_t option)
Reads the resource with data at locations passed in through parameters.

Verification

- status_t **FLEXNVM_DflashVerifyErase** (`flexnvm_config_t` *config, `uint32_t` start, `uint32_t` lengthInBytes, `ftfx_margin_value_t` margin)
Verifies an erasure of the desired flash area at a specified margin level.
- status_t **FLEXNVM_VerifyEraseAll** (`flexnvm_config_t` *config, `ftfx_margin_value_t` margin)
Verifies erasure of the entire flash at a specified margin level.
- status_t **FLEXNVM_DflashVerifyProgram** (`flexnvm_config_t` *config, `uint32_t` start, `uint32_t` lengthInBytes, `const uint8_t` *expectedData, `ftfx_margin_value_t` margin, `uint32_t` *failedAddress, `uint32_t` *failedData)
Verifies programming of the desired flash area at a specified margin level.

Security

- status_t **FLEXNVM_GetSecurityState** (`flexnvm_config_t` *config, `ftfx_security_state_t` *state)
Returns the security state via the pointer passed into the function.
- status_t **FLEXNVM_SecurityBypass** (`flexnvm_config_t` *config, `const uint8_t` *backdoorKey)
Allows users to bypass security with a backdoor key.

FlexRAM

- status_t **FLEXNVM_SetFlexramFunction** (`flexnvm_config_t` *config, `ftfx_flexram_func_opt_t` option)
Sets the FlexRAM function command.

Flash Protection Utilities

- status_t **FLEXNVM_DflashSetProtection** (`flexnvm_config_t` *config, `uint8_t` protectStatus)
Sets the DFlash protection to the intended protection status.
- status_t **FLEXNVM_DflashGetProtection** (`flexnvm_config_t` *config, `uint8_t` *protectStatus)
Gets the DFlash protection status.
- status_t **FLEXNVM_EepromSetProtection** (`flexnvm_config_t` *config, `uint8_t` protectStatus)
Sets the EEPROM protection to the intended protection status.
- status_t **FLEXNVM_EepromGetProtection** (`flexnvm_config_t` *config, `uint8_t` *protectStatus)
Gets the EEPROM protection status.

Properties

- status_t **FLEXNVMGetProperty** (`flexnvm_config_t` *config, `flexnvm_property_tag_t` whichProperty, `uint32_t` *value)
Returns the desired flexnvm property.

47.2 Data Structure Documentation

47.2.1 struct `flexnvm_config_t`

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

Function Documentation

47.3 Macro Definition Documentation

47.3.1 `#define FSL_FLEXNVM_DRIVER_VERSION (MAKE_VERSION(1, 0, 0))`

Version 1.0.0.

47.4 Enumeration Type Documentation

47.4.1 `enum flexnvm_property_tag_t`

Enumerator

`kFLEXNVM_PropertyDflashSectorSize` Dflash sector size property.

`kFLEXNVM_PropertyDflashTotalSize` Dflash total size property.

`kFLEXNVM_PropertyDflashBlockSize` Dflash block size property.

`kFLEXNVM_PropertyDflashBlockCount` Dflash block count property.

`kFLEXNVM_PropertyDflashBlockBaseAddr` Dflash block base address property.

`kFLEXNVM_PropertyFlexRamBlockBaseAddr` FlexRam block base address property.

`kFLEXNVM_PropertyFlexRamTotalSize` FlexRam total size property.

`kFLEXNVM_PropertyEepromTotalSize` EEPROM total size property.

47.5 Function Documentation

47.5.1 `status_t FLEXNVM_Init (flexnvm_config_t * config)`

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.

<code>#kStatus_FTFx_Partition- StatusUpdateFailure</code>	Failed to update the partition status.
---	--

47.5.2 **status_t FLEXNVM_DflashErase (*flexnvm_config_t * config, uint32_t start,* *uint32_t lengthInBytes, uint32_t key*)**

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

<i>config</i>	The pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
<i>key</i>	The value used to validate all flash erase APIs.

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_Invalid- Argument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_- AlignmentError</code>	The parameter is not aligned with the specified baseline.
<code>#kStatus_FTFx_Address- Error</code>	The address is out of range.
<code>#kStatus_FTFx_Erase- KeyError</code>	The API erase key is invalid.
<code>#kStatus_FTFx_Execute- InRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_Access- Error</code>	Invalid instruction codes and out-of bounds addresses.

Function Documentation

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

47.5.3 `status_t FLEXNVM_EraseAll(flexnvm_config_t * config, uint32_t key)`

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_EraseKeyError</code>	API erase key is invalid.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.
<code>#kStatus_FTFx_PartitionStatusUpdateFailure</code>	Failed to update the partition status.

47.5.4 `status_t FLEXNVM_DflashProgram(flexnvm_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)`

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with the specified baseline.
<code>#kStatus_FTFx_AddressError</code>	Address is out of range.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

47.5.5 `status_t FLEXNVM_DflashProgramSection (flexnvm_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)`

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

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_AddressError</code>	Address is out of range.
<code>#kStatus_FTFx_SetFlexramAsRamError</code>	Failed to set flexram as RAM.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.
<code>#kStatus_FTFx_RecoverFlexramAsEepromError</code>	Failed to recover FlexRAM as EEPROM.

47.5.6 `status_t FLEXNVM_ProgramPartition (flexnvm_config_t * config, ftfx_partition_flexram_load_opt_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)`

Parameters

<i>config</i>	Pointer to storage for the driver runtime state.
<i>option</i>	The option used to set FlexRAM load behavior during reset.
<i>eepromData-SizeCode</i>	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
<i>flexnvm-PartitionCode</i>	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	Invalid argument is provided.
#kStatus_FTFx_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.
#kStatus_FTFx_Access-Error	Invalid instruction codes and out-of bounds addresses.
#kStatus_FTFx_-ProtectionViolation	The program/erase operation is requested to execute on protected areas.
#kStatus_FTFx_-CommandFailure	Run-time error during command execution.

47.5.7 status_t FLEXNVM_ReadResource (flexnvm_config_t * *config*, uint32_t *start*, uint8_t * *dst*, uint32_t *lengthInBytes*, ftfx_read_resource_opt_t *option*)

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

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.

Function Documentation

<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

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with the specified baseline.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

47.5.8 `status_t FLEXNVM_DflashVerifyErase (flexnvm_config_t * config, uint32_t start, uint32_t lengthInBytes, ftfx_margin_value_t margin)`

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

Parameters

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

<i>margin</i>	Read margin choice.
---------------	---------------------

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_AddressError</code>	Address is out of range.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

47.5.9 `status_t FLEXNVM_VerifyEraseAll (flexnvm_config_t * config, ftfx_margin_value_t margin)`

This function checks whether the flash is erased to the specified read margin level.

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.

Function Documentation

<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

47.5.10 `status_t FLEXNVM_DflashVerifyProgram (flexnvm_config_t * config, uint32_t start, uint32_t lengthInBytes, const uint8_t * expectedData, ftfx_margin_value_t margin, uint32_t * failedAddress, uint32_t * failedData)`

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

Parameters

<code>config</code>	A pointer to the storage for the driver runtime state.
<code>start</code>	The start address of the desired flash memory to be verified. Must be word-aligned.
<code>lengthInBytes</code>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<code>expectedData</code>	A pointer to the expected data that is to be verified against.
<code>margin</code>	Read margin choice.
<code>failedAddress</code>	A pointer to the returned failing address.
<code>failedData</code>	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

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.

<code>#kStatus_FTFx_AlignmentError</code>	Parameter is not aligned with specified baseline.
<code>#kStatus_FTFx_AddressError</code>	Address is out of range.
<code>#kStatus_FTFx_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>#kStatus_FTFx_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

47.5.11 `status_t FLEXNVM_GetSecurityState (flexnvm_config_t * config, ftfx_security_state_t * state)`

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.

47.5.12 `status_t FLEXNVM_SecurityBypass (flexnvm_config_t * config, const uint8_t * backdoorKey)`

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Function Documentation

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>backdoorKey</i>	A pointer to the user buffer containing the backdoor key.

Return values

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

47.5.13 **status_t FLEXNVM_SetFlexramFunction (flexnvm_config_t * *config*, ftfx_flexram_func_opt_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

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_Invalid-Argument	An invalid argument is provided.
#kStatus_FTFx_Execute-InRamFunctionNotReady	Execute-in-RAM function is not available.

<code>#kStatus_FTFx_Error</code>	Invalid instruction codes and out-of bounds addresses.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during the command execution.

47.5.14 `status_t FLEXNVM_EepromWrite (flexnvm_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)`

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

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_AddressError</code>	Address is out of range.
<code>#kStatus_FTFx_SetFlexramAsEepromError</code>	Failed to set flexram as eeprom.
<code>#kStatus_FTFx_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.

Function Documentation

<code>#kStatus_FTFx_Recover-FlexramAsRamError</code>	Failed to recover the FlexRAM as RAM.
--	---------------------------------------

47.5.15 `status_t FLEXNVM_DflashSetProtection (flexnvm_config_t * config, uint8_t protectStatus)`

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_CommandNotSupported</code>	Flash API is not supported.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.

47.5.16 `status_t FLEXNVM_DflashGetProtection (flexnvm_config_t * config, uint8_t * protectStatus)`

Parameters

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

Return values

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_CommandNotSupported</code>	Flash API is not supported.

47.5.17 `status_t FLEXNVM_EepromSetProtection (flexnvm_config_t * config, uint8_t protectStatus)`

Parameters

<code>config</code>	A pointer to the storage for the driver runtime state.
<code>protectStatus</code>	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

<code>#kStatus_FTFx_Success</code>	API was executed successfully.
<code>#kStatus_FTFx_InvalidArgument</code>	An invalid argument is provided.
<code>#kStatus_FTFx_CommandNotSupported</code>	Flash API is not supported.
<code>#kStatus_FTFx_CommandFailure</code>	Run-time error during command execution.

47.5.18 `status_t FLEXNVM_EepromGetProtection (flexnvm_config_t * config, uint8_t * protectStatus)`

Parameters

Function Documentation

<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

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_InvalidArgument	An invalid argument is provided.
#kStatus_FTFx_CommandNotSupported	Flash API is not supported.

47.5.19 **status_t FLEXNVMGetProperty (flexnvm_config_t * config, flexnvm_property_tag_t whichProperty, uint32_t * value)**

Parameters

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

Return values

#kStatus_FTFx_Success	API was executed successfully.
#kStatus_FTFx_InvalidArgument	An invalid argument is provided.
#kStatus_FTFx_UnknownProperty	An unknown property tag.

Chapter 48

MMCCARD

48.1 Overview

Data Structures

- struct `mmccard_usr_param_t`
card user parameter, user can define the parameter according the board, card capability [More...](#)
- struct `mmc_card_t`
mmc card state [More...](#)

Enumerations

- enum `_mmc_card_flag` {
 `kMMC_SupportHighSpeed26MHZFlag` = (1U << 0U),
 `kMMC_SupportHighSpeed52MHZFlag` = (1U << 1U),
 `kMMC_SupportHighSpeedDDR52MHZ180V300VFlag` = (1 << 2U),
 `kMMC_SupportHighSpeedDDR52MHZ120VFlag` = (1 << 3U),
 `kMMC_SupportHS200200MHZ180VFlag` = (1 << 4U),
 `kMMC_SupportHS200200MHZ120VFlag` = (1 << 5U),
 `kMMC_SupportHS400DDR200MHZ180VFlag` = (1 << 6U),
 `kMMC_SupportHS400DDR200MHZ120VFlag` = (1 << 7U),
 `kMMC_SupportHighCapacityFlag` = (1U << 8U),
 `kMMC_SupportAlternateBootFlag` = (1U << 9U),
 `kMMC_SupportDDRBootFlag` = (1U << 10U),
 `kMMC_SupportHighSpeedBootFlag` = (1U << 11U) }
 MMC card flags.

MMCCARD Function

- status_t `MMC_Init` (`mmc_card_t` *card)
Initializes the MMC card and host.
- void `MMC_Deinit` (`mmc_card_t` *card)
Deinitializes the card and host.
- status_t `MMC_CardInit` (`mmc_card_t` *card)
initialize the card.
- void `MMC_CardDeinit` (`mmc_card_t` *card)
Deinitializes the card.
- status_t `MMC_HostInit` (`mmc_card_t` *card)
initialize the host.
- void `MMC_HostDeinit` (`mmc_card_t` *card)
Deinitializes the host.
- void `MMC_HostReset` (`SDMMCHOST_CONFIG` *host)
reset the host.
- void `MMC_PowerOnCard` (`SDMMCHOST_TYPE` *base, const `sdmmchost_pwr_card_t` *pwr)

Data Structure Documentation

- void **MMC_PowerOffCard** (SDMMCHOST_TYPE *base, const **sdmmchost_pwr_card_t** *pwr)
power off card.
- bool **MMC_CheckReadOnly** (**mmc_card_t** *card)
Checks if the card is read-only.
- status_t **MMC_ReadBlocks** (**mmc_card_t** *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)
Reads data blocks from the card.
- status_t **MMC_WriteBlocks** (**mmc_card_t** *card, const uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)
Writes data blocks to the card.
- status_t **MMC_EraseGroups** (**mmc_card_t** *card, uint32_t startGroup, uint32_t endGroup)
Erases groups of the card.
- status_t **MMC_SelectPartition** (**mmc_card_t** *card, mmc_access_partition_t partitionNumber)
Selects the partition to access.
- status_t **MMC_SetBootConfig** (**mmc_card_t** *card, const **mmc_boot_config_t** *config)
Configures the boot activity of the card.
- status_t **MMC_StartBoot** (**mmc_card_t** *card, const **mmc_boot_config_t** *mmcConfig, uint8_t *buffer, SDMMCHOST_BOOT_CONFIG *hostConfig)
MMC card start boot.
- status_t **MMC_SetBootConfigWP** (**mmc_card_t** *card, uint8_t wp)
MMC card set boot configuration write protect.
- status_t **MMC_ReadBootData** (**mmc_card_t** *card, uint8_t *buffer, SDMMCHOST_BOOT_CONFIG *hostConfig)
MMC card continous read boot data.
- status_t **MMC_StopBoot** (**mmc_card_t** *card, uint32_t bootMode)
MMC card stop boot mode.
- status_t **MMC_SetBootPartitionWP** (**mmc_card_t** *card, mmc_boot_partition_wp_t bootPartitionWP)
MMC card set boot partition write protect.

48.2 Data Structure Documentation

48.2.1 struct mmccard_usr_param_t

Data Fields

- const **sdmmchost_pwr_card_t** * pwr
power control configuration

48.2.2 struct mmc_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- SDMMCHOST_CONFIG host

Host information.

- **mmccard_usr_param_t usrParam**
 - user parameter*
- **bool isHostReady**
 - Use this flag to indicate if need host re-init or not.*
- **bool noInteralAlign**
 - use this flag to disable sdm mmc align.*
- **uint32_t busClock_Hz**
 - MMC bus clock united in Hz.*
- **uint32_t relativeAddress**
 - Relative address of the card.*
- **bool enablePreDefinedBlockCount**
 - Enable PRE-DEFINED block count when read/write.*
- **uint32_t flags**
 - Capability flag in _mmc_card_flag.*
- **uint32_t rawCid [4U]**
 - Raw CID content.*
- **uint32_t rawCsd [4U]**
 - Raw CSD content.*
- **uint32_t rawExtendedCsd [MMC_EXTENDED_CSD_BYTES/4U]**
 - Raw MMC Extended CSD content.*
- **uint32_t ocr**
 - Raw OCR content.*
- **mmc_cid_t cid**
 - CID.*
- **mmc_csd_t csd**
 - CSD.*
- **mmc_extended_csd_t extendedCsd**
 - Extended CSD.*
- **uint32_t blockSize**
 - Card block size.*
- **uint32_t userPartitionBlocks**
 - Card total block number in user partition.*
- **uint32_t bootPartitionBlocks**
 - Boot partition size united as block size.*
- **uint32_t eraseGroupBlocks**
 - Erase group size united as block size.*
- **mmc_access_partition_t currentPartition**
 - Current access partition.*
- **mmc_voltage_window_t hostVoltageWindowVCCQ**
 - Host IO voltage window.*
- **mmc_voltage_window_t hostVoltageWindowVCC**
 - application must set this value according to board specific*
- **mmc_high_speed_timing_t busTiming**
 - indicate the current work timing mode*
- **mmc_data_bus_width_t busWidth**
 - indicate the current work bus width*

Function Documentation

48.2.2.0.0.88 Field Documentation

48.2.2.0.0.88.1 bool mmc_card_t::noInternalAlign

If disable, sdmmc will not make sure the data buffer address is word align, otherwise all the transfer are align to low level driver

48.3 Enumeration Type Documentation

48.3.1 enum _mmc_card_flag

Enumerator

kMMC_SupportHighSpeed26MHZFlag Support high speed 26MHZ.

kMMC_SupportHighSpeed52MHZFlag Support high speed 52MHZ.

kMMC_SupportHighSpeedDDR52MHZ180V300VFlag ddr 52MHZ 1.8V or 3.0V

kMMC_SupportHighSpeedDDR52MHZ120VFlag DDR 52MHZ 1.2V.

kMMC_SupportHS200200MHZ180VFlag HS200 ,200MHZ,1.8V.

kMMC_SupportHS200200MHZ120VFlag HS200, 200MHZ, 1.2V.

kMMC_SupportHS400DDR200MHZ180VFlag HS400, DDR, 200MHZ,1.8V.

kMMC_SupportHS400DDR200MHZ120VFlag HS400, DDR, 200MHZ,1.2V.

kMMC_SupportHighCapacityFlag Support high capacity.

kMMC_SupportAlternateBootFlag Support alternate boot.

kMMC_SupportDDRBootFlag support DDR boot flag

kMMC_SupportHighSpeedBootFlag support high speed boot flag

48.4 Function Documentation

48.4.1 status_t MMC_Init (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_HostNotReady</i>	host is not ready.
<i>kStatus_SDMMC_GoIdleFailed</i>	Go idle failed.

<i>kStatus_SDMMC_SendOperationConditionFailed</i>	Send operation condition failed.
<i>kStatus_SDMMC_AllSendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_SetRelativeAddressFailed</i>	Set relative address failed.
<i>kStatus_SDMMC_SendCsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_CardNotSupport</i>	Card not support.
<i>kStatus_SDMMC_SelectCardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_SendExtendedCsdFailed</i>	Send EXT_CSD failed.
<i>kStatus_SDMMC_SetBusWidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_SwitchHighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_SetCardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.2 void MMC_Deinit (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

48.4.3 status_t MMC_CardInit (mmc_card_t * *card*)

Parameters

Function Documentation

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Host-NotReady</i>	host is not ready.
<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Set-RelativeAddressFailed</i>	Set relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ExtendedCsdFailed</i>	Send EXT_CSD failed.
<i>kStatus_SDMMC_SetBus-WidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_Switch-HighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.4 void MMC_CardDeinit (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

48.4.5 **status_t MMC_HostInit (mmc_card_t * *card*)**

This function deinitializes the specific host.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

48.4.6 **void MMC_HostDeinit (mmc_card_t * *card*)**

This function deinitializes the host.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

48.4.7 **void MMC_HostReset (SDMMCHOST_CONFIG * *host*)**

This function reset the specific host.

Parameters

<i>host</i>	host descriptor.
-------------	------------------

48.4.8 **void MMC_PowerOnCard (SDMMCHOST_TYPE * *base*, const sdmmchost_pwr_card_t * *pwr*)**

The power on operation depend on host or the user define power on function.

Parameters

<i>base</i>	host base address.
-------------	--------------------

Function Documentation

<i>pwr</i>	user define power control configuration
------------	---

48.4.9 void MMC_PowerOffCard (SDMMCHOST_TYPE * *base*, const sdmmchost_pwr_card_t * *pwr*)

The power off operation depend on host or the user define power on function.

Parameters

<i>base</i>	host base address.
<i>pwr</i>	user define power control configuration

48.4.10 bool MMC_CheckReadOnly (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

48.4.11 status_t MMC_ReadBlocks (mmc_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save data.
<i>startBlock</i>	The start block index.

<i>blockCount</i>	The number of blocks to read.
-------------------	-------------------------------

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Set-BlockCountFailed</i>	Set block count failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.12 **status_t MMC_WriteBlocks (mmc_card_t * *card*, const uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)**

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save data blocks.
<i>startBlock</i>	Start block number to write.
<i>blockCount</i>	Block count.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Set-BlockCountFailed</i>	Set block count failed.

Function Documentation

<i>kStatus_SDMMC_WaitWriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_StopTransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.13 **status_t MMC_EraseGroups (mmc_card_t * card, uint32_t startGroup, uint32_t endGroup)**

Erase group is the smallest erase unit in MMC card. The erase range is [startGroup, endGroup].

Parameters

<i>card</i>	Card descriptor.
<i>startGroup</i>	Start group number.
<i>endGroup</i>	End group number.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_WaitWriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.14 **status_t MMC_SelectPartition (mmc_card_t * card, mmc_access_partition_t partitionNumber)**

Parameters

<i>card</i>	Card descriptor.
<i>partition-Number</i>	The partition number.

Return values

<i>kStatus_SDMMC_ConfigureExtendedCsdFailed</i>	Configure EXT_CSD failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.15 **status_t MMC_SetBootConfig (mmc_card_t * *card*, const mmc_boot_config_t * *config*)**

Parameters

<i>card</i>	Card descriptor.
<i>config</i>	Boot configuration structure.

Return values

<i>kStatus_SDMMC_NotSupportYet</i>	Not support now.
<i>kStatus_SDMMC_ConfigureExtendedCsdFailed</i>	Configure EXT_CSD failed.
<i>kStatus_SDMMC_ConfigureBootFailed</i>	Configure boot failed.
<i>kStatus_Success</i>	Operate successfully.

48.4.16 **status_t MMC_StartBoot (mmc_card_t * *card*, const mmc_boot_config_t * *mmcConfig*, uint8_t * *buffer*, SDMMCHOST_BOOT_CONFIG * *hostConfig*)**

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>mmcConfig</i>	mmc Boot configuration structure.
<i>buffer</i>	address to receive data.
<i>hostConfig</i>	host boot configurations.

Return values

<i>kStatus_Fail</i>	fail.
<i>kStatus_SDMMC_-TransferFailed</i>	transfer fail.
<i>kStatus_SDMMC_Go-IdleFailed</i>	reset card fail.
<i>kStatus_Success</i>	Operate successfully.

48.4.17 **status_t MMC_SetBootConfigWP (mmc_card_t * *card*, uint8_t *wp*)**

Parameters

<i>card</i>	Card descriptor.
<i>wp</i>	write protect value.

48.4.18 **status_t MMC_ReadBootData (mmc_card_t * *card*, uint8_t * *buffer*, SDMMCHOST_BOOT_CONFIG * *hostConfig*)**

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	buffer address.
<i>hostConfig</i>	host boot configurations.

48.4.19 **status_t MMC_StopBoot (mmc_card_t * *card*, uint32_t *bootMode*)**

Parameters

<i>card</i>	Card descriptor.
<i>bootMode</i>	boot mode.

48.4.20 **status_t MMC_SetBootPartitionWP (mmc_card_t * *card*, mmc_boot_partition_wp_t *bootPartitionWP*)**

Parameters

<i>card</i>	Card descriptor.
<i>bootPartitionWP</i>	boot partition write protect value.

Function Documentation

Chapter 49

SDCARD

49.1 Overview

Data Structures

- struct `sdcard_usr_param_t`
card user parameter, user can define the parameter according the board, card capability [More...](#)
- struct `sd_card_t`
SD card state. [More...](#)

Enumerations

- enum `_sd_card_flag` {
 `kSD_SupportHighCapacityFlag` = (1U << 1U),
 `kSD_Support4BitWidthFlag` = (1U << 2U),
 `kSD_SupportSdhcFlag` = (1U << 3U),
 `kSD_SupportSdxcFlag` = (1U << 4U),
 `kSD_SupportVoltage180v` = (1U << 5U),
 `kSD_SupportSetBlockCountCmd` = (1U << 6U),
 `kSD_SupportSpeedClassControlCmd` = (1U << 7U) }
SD card flags.

SDCARD Function

- `status_t SD_Init (sd_card_t *card)`
Initializes the card on a specific host controller.
- `void SD_Deinit (sd_card_t *card)`
Deinitializes the card.
- `status_t SD_CardInit (sd_card_t *card)`
Initializes the card.
- `void SD_CardDeinit (sd_card_t *card)`
Deinitializes the card.
- `status_t SD_HostInit (sd_card_t *card)`
initialize the host.
- `void SD_HostDeinit (sd_card_t *card)`
Deinitializes the host.
- `void SD_HostReset (SDMMCHOST_CONFIG *host)`
reset the host.
- `void SD_PowerOnCard (SDMMCHOST_TYPE *base, const sdmmchost_pwr_card_t *pwr)`
power on card.
- `void SD_PowerOffCard (SDMMCHOST_TYPE *base, const sdmmchost_pwr_card_t *pwr)`
power off card.
- `status_t SD_WaitCardDetectStatus (SDMMCHOST_TYPE *hostBase, const sdmmchost_detect_card_t *cd, bool waitCardStatus)`

Data Structure Documentation

- bool **SD_IsCardPresent** (*sd_card_t* *card)
sd card present check function.
- bool **SD_CheckReadOnly** (*sd_card_t* *card)
Checks whether the card is write-protected.
- status_t **SD_SelectCard** (*sd_card_t* *card, bool isSelected)
Send SELECT_CARD command to set the card to be transfer state or not.
- status_t **SD_ReadStatus** (*sd_card_t* *card)
Send ACMD13 to get the card current status.
- status_t **SD_ReadBlocks** (*sd_card_t* *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)
Reads blocks from the specific card.
- status_t **SD_WriteBlocks** (*sd_card_t* *card, const uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)
Writes blocks of data to the specific card.
- status_t **SD_EraseBlocks** (*sd_card_t* *card, uint32_t startBlock, uint32_t blockCount)
Erases blocks of the specific card.
- status_t **SD_SetDriverStrength** (*sd_card_t* *card, sd_driver_strength_t driverStrength)
select card driver strength select card driver strength
- status_t **SD_SetMaxCurrent** (*sd_card_t* *card, sd_max_current_t maxCurrent)
select max current select max operation current

49.2 Data Structure Documentation

49.2.1 struct sdcards_usr_param_t

Data Fields

- const **sdcards_detect_card_t** * cd
card detect type
- const **sdcards_pwr_card_t** * pwr
power control configuration

49.2.2 struct sd_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- SDMMCHOST_CONFIG host
Host information.
- **sdcards_usr_param_t** usrParam
user parameter
- bool **isHostReady**
use this flag to indicate if need host re-init or not
- bool **noInternalAlign**
use this flag to disable sdmmc align.

- `uint32_t busClock_Hz`
SD bus clock frequency united in Hz.
- `uint32_t relativeAddress`
Relative address of the card.
- `uint32_t version`
Card version.
- `uint32_t flags`
Flags in _sd_card_flag.
- `uint32_t rawCid [4U]`
Raw CID content.
- `uint32_t rawCsd [4U]`
Raw CSD content.
- `uint32_t rawScr [2U]`
Raw CSD content.
- `uint32_t ocr`
Raw OCR content.
- `sd_cid_t cid`
CID.
- `sd_csd_t csd`
CSD.
- `sd_scr_t scr`
SCR.
- `sd_status_t stat`
sd 512 bit status
- `uint32_t blockCount`
Card total block number.
- `uint32_t blockSize`
Card block size.
- `sd_timing_mode_t currentTiming`
current timing mode
- `sd_driver_strength_t driverStrength`
driver strength
- `sd_max_current_t maxCurrent`
card current limit
- `sdmmc_operation_voltage_t operationVoltage`
card operation voltage

49.2.2.0.0.89 Field Documentation

49.2.2.0.0.89.1 `bool sd_card_t::noInternalAlign`

If disable, sdmmc will not make sure the data buffer address is word align, otherwise all the transfer are align to low level driver

49.3 Enumeration Type Documentation

49.3.1 `enum _sd_card_flag`

Enumerator

`kSD_SupportHighCapacityFlag` Support high capacity.

Function Documentation

kSD_Support4BitWidthFlag Support 4-bit data width.

kSD_SupportSdhcFlag Card is SDHC.

kSD_SupportSdxcFlag Card is SDXC.

kSD_SupportVoltage180v card support 1.8v voltage

kSD_SupportSetBlockCountCmd card support cmd23 flag

kSD_SupportSpeedClassControlCmd card support speed class control flag

49.4 Function Documentation

49.4.1 status_t SD_Init (sd_card_t * card)

This function initializes the card on a specific host controller, it is consist of host init, card detect, card init function, however user can ignore this high level function, instead of use the low level function, such as SD_CardInit, SD_HostInit, SD_CardDetect.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Host-NotReady</i>	host is not ready.
<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Not-SupportYet</i>	Card not support.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Send-RelativeAddressFailed</i>	Send relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.

<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ScrFailed</i>	Send SCR failed.
<i>kStatus_SDMMC_SetBus-WidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_Switch-HighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.2 void SD_Deinit (*sd_card_t * card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

49.4.3 status_t SD_CardInit (*sd_card_t * card*)

This function initializes the card only, make sure the host is ready when call this function, otherwise it will return *kStatus_SDMMC_HostNotReady*.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Host-NotReady</i>	host is not ready.
<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.

Function Documentation

<i>kStatus_SDMMC_NotSupportYet</i>	Card not support.
<i>kStatus_SDMMC_SendOperationConditionFailed</i>	Send operation condition failed.
<i>kStatus_SDMMC_AllSendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_SendRelativeAddressFailed</i>	Send relative address failed.
<i>kStatus_SDMMC_SendCsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_SelectCardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_SendScrFailed</i>	Send SCR failed.
<i>kStatus_SDMMC_SetBusWidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_SwitchHighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_SetCardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.4 void SD_CardDeinit (*sd_card_t * card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

49.4.5 status_t SD_HostInit (*sd_card_t * card*)

This function deinitializes the specific host.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

49.4.6 void SD_HostDeinit (*sd_card_t* * *card*)

This function deinitializes the host.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

49.4.7 void SD_HostReset (*SDMMCHOST_CONFIG* * *host*)

This function reset the specific host.

Parameters

<i>host</i>	host descriptor.
-------------	------------------

49.4.8 void SD_PowerOnCard (*SDMMCHOST_TYPE* * *base*, *const sdmmchost_pwr_card_t* * *pwr*)

The power on operation depend on host or the user define power on function.

Parameters

<i>base</i>	host base address.
<i>pwr</i>	user define power control configuration

49.4.9 void SD_PowerOffCard (*SDMMCHOST_TYPE* * *base*, *const sdmmchost_pwr_card_t* * *pwr*)

The power off operation depend on host or the user define power on function.

Function Documentation

Parameters

<i>base</i>	host base address.
<i>pwr</i>	user define power control configuration

49.4.10 **status_t SD_WaitCardDetectStatus (SDMMCHOST_TYPE * *hostBase*, const sdmmchost_detect_card_t * *cd*, bool *waitCardStatus*)**

Detect card through GPIO, CD, DATA3.

Parameters

<i>card</i>	card descriptor.
<i>card</i>	detect configuration
<i>waitCardStatus</i>	wait card detect status

49.4.11 **bool SD_IsCardPresent (sd_card_t * *card*)**

Parameters

<i>card</i>	card descriptor.
-------------	------------------

49.4.12 **bool SD_CheckReadOnly (sd_card_t * *card*)**

This function checks if the card is write-protected via the CSD register.

Parameters

<i>card</i>	The specific card.
-------------	--------------------

Return values

<i>true</i>	Card is read only.
-------------	--------------------

<i>false</i>	Card isn't read only.
--------------	-----------------------

49.4.13 status_t SD_SelectCard (sd_card_t * *card*, bool *isSelected*)

Parameters

<i>card</i>	Card descriptor.
<i>isSelected</i>	True to set the card into transfer state, false to disselect.

Return values

<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.14 status_t SD_ReadStatus (sd_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_SendApplicationCommandFailed</i>	send application command failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.15 status_t SD_ReadBlocks (sd_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

This function reads blocks from the specific card with default block size defined by the SDHC_CARD_DEFAULT_BLOCK_SIZE.

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save the data read from card.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to read.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.16 **status_t SD_WriteBlocks (sd_card_t * card, const uint8_t * buffer, uint32_t startBlock, uint32_t blockCount)**

This function writes blocks to the specific card with default block size 512 bytes.

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer holding the data to be written to the card.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to write.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_NotSupportYet</i>	Not support now.
<i>kStatus_SDMMC_CardNotSupport</i>	Card not support.
<i>kStatus_SDMMC_WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_StopTransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.17 **status_t SD_EraseBlocks (sd_card_t * card, uint32_t startBlock, uint32_t blockCount)**

This function erases blocks of the specific card with default block size 512 bytes.

Parameters

<i>card</i>	Card descriptor.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to erase.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.

Function Documentation

<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_Success</i>	Operate successfully.

49.4.18 **status_t SD_SetDriverStrength (*sd_card_t * card, sd_driver_strength_t driverStrength*)**

Parameters

<i>card</i>	Card descriptor.
<i>driverStrength</i>	Driver strength

49.4.19 **status_t SD_SetMaxCurrent (*sd_card_t * card, sd_max_current_t maxCurrent*)**

Parameters

<i>card</i>	Card descriptor.
<i>maxCurrent</i>	Max current

Chapter 50

SDIOCARD

50.1 Overview

Data Structures

- struct `sdiocard_usr_param_t`
card user parameter, user can define the parameter according the board, card capability [More...](#)
- struct `sdio_card_t`
SDIO card state. [More...](#)

Macros

- #define `FSL_SDIO_DRIVER_VERSION` (MAKE_VERSION(2U, 2U, 6U)) /*2.2.6*/
Middleware version.

SDIOCARD Function

- status_t `SDIO_Init` (`sdio_card_t` *card)
SDIO card init function.
- void `SDIO_Deinit` (`sdio_card_t` *card)
SDIO card_deinit, include card and host_deinit.
- status_t `SDIO_CardInit` (`sdio_card_t` *card)
Initializes the card.
- void `SDIO_CardDeinit` (`sdio_card_t` *card)
Deinitializes the card.
- status_t `SDIO_HostInit` (`sdio_card_t` *card)
initialize the host.
- void `SDIO_HostDeinit` (`sdio_card_t` *card)
Deinitializes the host.
- void `SDIO_HostReset` (`SDMMCHOST_CONFIG` *host)
reset the host.
- void `SDIO_PowerOnCard` (`SDMMCHOST_TYPE` *base, const `sdmmchost_pwr_card_t` *pwr)
power on card.
- void `SDIO_PowerOffCard` (`SDMMCHOST_TYPE` *base, const `sdmmchost_pwr_card_t` *pwr)
power on card.
- status_t `SDIO_CardInActive` (`sdio_card_t` *card)
set SDIO card to inactive state
- status_t `SDIO_IO_Write_Direct` (`sdio_card_t` *card, `sdio_func_num_t` func, `uint32_t` regAddr, `uint8_t` *data, `bool` raw)
IO direct write transfer function.
- status_t `SDIO_IO_Read_Direct` (`sdio_card_t` *card, `sdio_func_num_t` func, `uint32_t` regAddr, `uint8_t` *data)
IO direct read transfer function.
- status_t `SDIO_IO_Write_Extended` (`sdio_card_t` *card, `sdio_func_num_t` func, `uint32_t` regAddr, `uint8_t` *buffer, `uint32_t` count, `uint32_t` flags)

Data Structure Documentation

- status_t **SDIO_IO_Read_Extended** (sdio_card_t *card, sdio_func_num_t func, uint32_t regAddr, uint8_t *buffer, uint32_t count, uint32_t flags)
IO extended write transfer function.
- status_t **SDIO_GetCardCapability** (sdio_card_t *card, sdio_func_num_t func)
get SDIO card capability
- status_t **SDIO_SetBlockSize** (sdio_card_t *card, sdio_func_num_t func, uint32_t blockSize)
set SDIO card block size
- status_t **SDIO_CardReset** (sdio_card_t *card)
set SDIO card reset
- status_t **SDIO_SetDataBusWidth** (sdio_card_t *card, sdio_bus_width_t busWidth)
set SDIO card data bus width
- status_t **SDIO_SwitchToHighSpeed** (sdio_card_t *card)
switch the card to high speed
- status_t **SDIO_ReadCIS** (sdio_card_t *card, sdio_func_num_t func, const uint32_t *tupleList, uint32_t tupleNum)
read SDIO card CIS for each function
- status_t **SDIO_EnableIOInterrupt** (sdio_card_t *card, sdio_func_num_t func, bool enable)
enable IO interrupt
- status_t **SDIO_EnableIO** (sdio_card_t *card, sdio_func_num_t func, bool enable)
enable IO and wait IO ready
- status_t **SDIO_SelectIO** (sdio_card_t *card, sdio_func_num_t func)
select IO
- status_t **SDIO_AbortIO** (sdio_card_t *card, sdio_func_num_t func)
Abort IO transfer
- status_t **SDIO_WaitCardDetectStatus** (SDMMCHOST_TYPE *hostBase, const sdmmchost_detect_card_t *cd, bool waitCardStatus)
sdio wait card detect function.
- bool **SDIO_IsCardPresent** (sdio_card_t *card)
sdio card present check function.
- status_t **SDIO_IO_Transfer** (sdio_card_t *card, sdio_command_t cmd, uint32_t argument, uint32_t blockSize, uint8_t *txData, uint8_t *rxData, uint16_t dataSize, uint32_t *response)
sdio card io transfer function.

50.2 Data Structure Documentation

50.2.1 struct sdiocard_usr_param_t

Data Fields

- const sdmmchost_detect_card_t * cd
card detect type
- const sdmmchost_pwr_card_t * pwr
power control configuration

50.2.2 struct sdio_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- **SDMMCHOST_CONFIG host**
Host information.
- **sdiocard_usr_param_t usrParam**
user parameter
- **bool isHostReady**
use this flag to indicate if need host re-init or not
- **bool memPresentFlag**
indicate if memory present
- **uint32_t busClock_Hz**
SD bus clock frequency united in Hz.
- **uint32_t relativeAddress**
Relative address of the card.
- **uint8_t sdVersion**
SD version.
- **uint8_t sdioVersion**
SDIO version.
- **uint8_t cccrVersioin**
CCCR version.
- **uint8_t ioTotalNumber**
total number of IO function
- **uint32_t cccrflags**
Flags in _sd_card_flag.
- **uint32_t io0blockSize**
record the io0 block size
- **uint32_t ocr**
Raw OCR content, only 24bit available for SDIO card.
- **uint32_t commonCISPointer**
point to common CIS
- **sdio_fbr_t ioFBR [7U]**
FBR table.
- **sdio_common_cis_t commonCIS**
CIS table.
- **sdio_func_cis_t funcCIS [7U]**
function CIS table

50.3 Macro Definition Documentation

#define FSL_SDIO_DRIVER_VERSION (MAKE_VERSION(2U, 2U, 6U))
/*2.2.6*/

50.4 Function Documentation

status_t SDIO_Init (sdio_card_t * card)

Function Documentation

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Go-IdleFailed</i>	
<i>kStatus_SDMMC_Hand-ShakeOperation-ConditionFailed</i>	
<i>kStatus_SDMMC_SDIO-InvalidCard</i>	
<i>kStatus_SDMMC_SDIO-InvalidVoltage</i>	
<i>kStatus_SDMMC_Send-RelativeAddressFailed</i>	
<i>kStatus_SDMMC_Select-CardFailed</i>	
<i>kStatus_SDMMC_SDIO-SwitchHighSpeedFail</i>	
<i>kStatus_SDMMC_SDIO-ReadCISFail</i>	
<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.2 void SDIO_Deinit (*sdio_card_t * card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

50.4.3 status_t SDIO_CardInit (*sdio_card_t * card*)

This function initializes the card only, make sure the host is ready when call this function, otherwise it will return *kStatus_SDMMC_HostNotReady*.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Host-NotReady</i>	host is not ready.
<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Not-SupportYet</i>	Card not support.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Send-RelativeAddressFailed</i>	Send relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ScrFailed</i>	Send SCR failed.
<i>kStatus_SDMMC_SetBus-WidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_Switch-HighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

50.4.4 void SDIO_CardDeinit (sdio_card_t * *card*)

This function deinitializes the specific card.

Function Documentation

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

50.4.5 **status_t SDIO_HostInit (*sdio_card_t* * *card*)**

This function deinitializes the specific host.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

50.4.6 **void SDIO_HostDeinit (*sdio_card_t* * *card*)**

This function deinitializes the host.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

50.4.7 **void SDIO_HostReset (*SDMMCHOST_CONFIG* * *host*)**

This function reset the specific host.

Parameters

<i>host</i>	host descriptor.
-------------	------------------

50.4.8 **void SDIO_PowerOnCard (*SDMMCHOST_TYPE* * *base*, *const sdmmchost_pwr_card_t* * *pwr*)**

The power on operation depend on host or the user define power on function.

Parameters

<i>base</i>	host base address.
<i>pwr</i>	user define power control configuration

50.4.9 void SDIO_PowerOffCard (**SDMMCHOST_TYPE** * *base*, const **sdmmchost_pwr_card_t** * *pwr*)

The power off operation depend on host or the user define power on function.

Parameters

<i>base</i>	host base address.
<i>pwr</i>	user define power control configuration

50.4.10 status_t SDIO_CardInActive (**sdio_card_t** * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.11 status_t SDIO_IO_Write_Direct (**sdio_card_t** * *card*, **sdio_func_num_t** *func*, **uint32_t** *regAddr*, **uint8_t** * *data*, **bool** *raw*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO numner

Function Documentation

<i>register</i>	address
<i>the</i>	data pointer to write
<i>raw</i>	flag, indicate read after write or write only

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.12 **status_t SDIO_IO_Read_Direct (*sdio_card_t * card, sdio_func_num_t func, uint32_t regAddr, uint8_t * data*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	pointer to read

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.13 **status_t SDIO_IO_Write_Extended (*sdio_card_t * card, sdio_func_num_t func, uint32_t regAddr, uint8_t * buffer, uint32_t count, uint32_t flags*)**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	buffer to write
<i>data</i>	count
<i>write</i>	flags

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidArgument</i>	
<i>kStatus_Success</i>	

50.4.14 **status_t SDIO_IO_Read_Extended (*sdio_card_t * card, sdio_func_num_t func, uint32_t regAddr, uint8_t * buffer, uint32_t count, uint32_t flags*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	buffer to read
<i>data</i>	count
<i>write</i>	flags

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidArgument</i>	

Function Documentation

<i>kStatus_Success</i>

50.4.15 **status_t SDIO_GetCardCapability (*sdio_card_t * card, sdio_func_num_t func*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.16 **status_t SDIO_SetBlockSize (*sdio_card_t * card, sdio_func_num_t func, uint32_t blockSize*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	io number
<i>block</i>	size

Return values

<i>kStatus_SDMMC_SetCardBlockSizeFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidArgument</i>	

<i>kStatus_Success</i>

50.4.17 **status_t SDIO_CardReset (*sdio_card_t * card*)**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.18 **status_t SDIO_SetDataBusWidth (*sdio_card_t * card, sdio_bus_width_t busWidth*)**

Parameters

<i>card</i>	Card descriptor.
<i>data</i>	bus width

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.19 **status_t SDIO_SwitchToHighSpeed (*sdio_card_t * card*)**

Parameters

Function Documentation

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_SDMMC_SDIO_SwitchHighSpeedFail</i>	
<i>kStatus_Success</i>	

50.4.20 status_t SDIO_ReadCIS (*sdio_card_t * card, sdio_func_num_t func, const uint32_t * tupleList, uint32_t tupleNum*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	io number
<i>tuple</i>	code list
<i>tuple</i>	code number

Return values

<i>kStatus_SDMMC_SDIO_ReadCISFail</i>	
<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.21 status_t SDIO_EnableIOInterrupt (*sdio_card_t * card, sdio_func_num_t func, bool enable*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>enable/disable</i>	flag

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.22 **status_t SDIO_EnableIO (*sdio_card_t * card, sdio_func_num_t func, bool enable*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>enable/disable</i>	flag

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.23 **status_t SDIO_SelectIO (*sdio_card_t * card, sdio_func_num_t func*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

Function Documentation

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.24 **status_t SDIO_AbortIO (*sdio_card_t* * *card*, *sdio_func_num_t* *func*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

50.4.25 **status_t SDIO_WaitCardDetectStatus (*SDMMCHOST_TYPE* * *hostBase*, *const sdmmchost_detect_card_t* * *cd*, *bool* *waitCardStatus*)**

Detect card through GPIO, CD, DATA3.

Parameters

<i>card</i>	card descriptor.
<i>card</i>	detect configuration
<i>waitCardStatus</i>	wait card detect status

50.4.26 **bool SDIO_IsCardPresent (*sdio_card_t* * *card*)**

Parameters

<i>card</i>	card descriptor.
-------------	------------------

50.4.27 `status_t SDIO_IO_Transfer (sdio_card_t * card, sdio_command_t cmd, uint32_t argument, uint32_t blockSize, uint8_t * txData, uint8_t * rxData, uint16_t dataSize, uint32_t * response)`

This function can be used for transfer direct/extend command. Please pay attention to the non-align data buffer address transfer, if data buffer address can not meet host controller internal DMA requirement, sdio driver will try to use internal align buffer if data size is not bigger than internal buffer size, Align address transfer always can get a better performance, so if application want sdio driver make sure buffer address align, please redefine the SDMMC_GLOBAL_BUFFER_SIZE macro to a value which is big enough for your application.

Parameters

<i>card</i>	card descriptor.
<i>cmd</i>	command to transfer
<i>argument</i>	argument to transfer
<i>blockSize</i>	used for block mode.
<i>txData</i>	tx buffer pointer or NULL
<i>rxData</i>	rx buffer pointer or NULL
<i>dataSize</i>	transfer data size
<i>response</i>	reponse pointer, if application want read response back, please set it to a NON-NULL pointer.

Function Documentation

Chapter 51

Data Structure Documentation

51.0.28 ftxf_config_t Struct Reference

Flash driver state information.

```
#include <fsl_ftfx_controller.h>
```

Data Fields

- `uint32_t flexramBlockBase`
The base address of the FlexRAM/acceleration RAM.
- `uint32_t flexramTotalSize`
The size of the FlexRAM/acceleration RAM.
- `uint16_t eepromTotalSize`
The size of EEPROM area which was partitioned from FlexRAM.
- `uint32_t * runCmdFuncAddr`
An buffer point to the flash execute-in-RAM function.

51.0.28.1 Detailed Description

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

51.0.28.2 Field Documentation

51.0.28.2.1 `uint32_t* ftxf_config_t::runCmdFuncAddr`

51.0.29 ftxf_ifr_desc_t Struct Reference

Flash IFR memory descriptor.

```
#include <fsl_ftfx_controller.h>
```

51.0.30 ftxf_mem_desc_t Struct Reference

Flash memory descriptor.

```
#include <fsl_ftfx_controller.h>
```

Data Fields

- `uint8_t type`
Type of flash block.
- `uint8_t index`
Index of flash block.
- `uint32_t blockBase`
A base address of the flash block.
- `uint32_t totalSize`
The size of the flash block.
- `uint32_t sectorSize`
The size in bytes of a sector of flash.
- `uint32_t blockCount`
A number of flash blocks.

51.0.30.1 Field Documentation

51.0.30.1.1 `uint8_t ftfx_mem_desc_t::type`

51.0.30.1.2 `uint8_t ftfx_mem_desc_t::index`

51.0.30.1.3 `uint32_t ftfx_mem_desc_t::totalSize`

51.0.30.1.4 `uint32_t ftfx_mem_desc_t::sectorSize`

51.0.30.1.5 `uint32_t ftfx_mem_desc_t::blockCount`

51.0.31 ftfx_ops_config_t Struct Reference

Active FTFx information for the current operation.

```
#include <fsl_ftfx_controller.h>
```

Data Fields

- `uint32_t convertedAddress`
A converted address for the current flash type.

51.0.31.1 Field Documentation

51.0.31.1.1 `uint32_t ftfx_ops_config_t::convertedAddress`

51.0.32 ftfx_spec_mem_t Struct Reference

ftfx special memory access information.

```
#include <fsl_ftfx_controller.h>
```

Data Fields

- `uint32_t base`
Base address of flash special memory.
- `uint32_t size`
size of flash special memory.
- `uint32_t count`
flash special memory count.

51.0.32.1 Field Documentation

51.0.32.1.1 `uint32_t ftfx_spec_mem_t::base`

51.0.32.1.2 `uint32_t ftfx_spec_mem_t::size`

51.0.32.1.3 `uint32_t ftfx_spec_mem_t::count`

51.0.33 `ftfx_swap_state_config_t` Struct Reference

Flash Swap information.

```
#include <fsl_ftfx_controller.h>
```

Data Fields

- `ftfx_swap_state_t flashSwapState`
The current Swap system status.
- `ftfx_swap_block_status_t currentSwapBlockStatus`
The current Swap block status.
- `ftfx_swap_block_status_t nextSwapBlockStatus`
The next Swap block status.

51.0.33.1 Field Documentation

51.0.33.1.1 `ftfx_swap_state_t ftfx_swap_state_config_t::flashSwapState`

51.0.33.1.2 `ftfx_swap_block_status_t ftfx_swap_state_config_t::currentSwapBlockStatus`

51.0.33.1.3 `ftfx_swap_block_status_t ftfx_swap_state_config_t::nextSwapBlockStatus`

51.0.34 `mmc_boot_config_t` Struct Reference

MMC card boot configuration definition.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- bool **enableBootAck**
Enable boot ACK.
- mmc_boot_partition_enable_t **bootPartition**
Boot partition.
- mmc_boot_timing_mode_t **bootTimingMode**
boot mode
- mmc_data_bus_width_t **bootDataBusWidth**
Boot data bus width.
- bool **retainBootbusCondition**
If retain boot bus width and boot mode conditions.
- bool **pwrBootConfigProtection**
Disable the change of boot configuration register bits from at this point until next power cycle or next H/W reset operation
- bool **permBootConfigProtection**
Disable the change of boot configuration register bits permanently.
- mmc_boot_partition_wp_t **bootPartitionWP**
boot partition write protect configurations

51.0.34.1 Detailed Description

51.0.35 mmc_cid_t Struct Reference

MMC card CID register.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- uint8_t **manufacturerID**
Manufacturer ID.
- uint16_t **applicationID**
OEM/Application ID.
- uint8_t **productName** [MMC_PRODUCT_NAME_BYTES]
Product name.
- uint8_t **productVersion**
Product revision.
- uint32_t **productSerialNumber**
Product serial number.
- uint8_t **manufacturerData**
Manufacturing date.

51.0.35.1 Detailed Description

51.0.36 mmc_csd_t Struct Reference

MMC card CSD register.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- `uint8_t csdStructureVersion`
CSD structure [127:126].
- `uint8_t systemSpecificationVersion`
System specification version [125:122].
- `uint8_t dataReadAccessTime1`
Data read access-time 1 [119:112].
- `uint8_t dataReadAccessTime2`
*Data read access-time 2 in CLOCK cycles (NSAC*100) [111:104].*
- `uint8_t transferSpeed`
Max.
- `uint16_t cardCommandClass`
card command classes [95:84]
- `uint8_t readBlockLength`
Max.
- `uint16_t flags`
Contain flags in _mmc_csd_flag.
- `uint16_t deviceSize`
Device size [73:62].
- `uint8_t readCurrentVddMin`
Max.
- `uint8_t readCurrentVddMax`
Max.
- `uint8_t writeCurrentVddMin`
Max.
- `uint8_t writeCurrentVddMax`
Max.
- `uint8_t deviceSizeMultiplier`
Device size multiplier [49:47].
- `uint8_t eraseGroupSize`
Erase group size [46:42].
- `uint8_t eraseGroupSizeMultiplier`
Erase group size multiplier [41:37].
- `uint8_t writeProtectGroupSize`
Write protect group size [36:32].
- `uint8_t defaultEcc`
Manufacturer default ECC [30:29].
- `uint8_t writeSpeedFactor`
Write speed factor [28:26].
- `uint8_t maxWriteBlockLength`
Max.

- `uint8_t fileFormat`
File format [11:10].
- `uint8_t eccCode`
ECC code [9:8].

51.0.36.1 Detailed Description

51.0.36.2 Field Documentation

51.0.36.2.1 `uint8_t mmc_csd_t::transferSpeed`

bus clock frequency [103:96]

51.0.36.2.2 `uint8_t mmc_csd_t::readBlockLength`

read data block length [83:80]

51.0.36.2.3 `uint8_t mmc_csd_t::readCurrentVddMin`

read current @ VDD min [61:59]

51.0.36.2.4 `uint8_t mmc_csd_t::readCurrentVddMax`

read current @ VDD max [58:56]

51.0.36.2.5 `uint8_t mmc_csd_t::writeCurrentVddMin`

write current @ VDD min [55:53]

51.0.36.2.6 `uint8_t mmc_csd_t::writeCurrentVddMax`

write current @ VDD max [52:50]

51.0.36.2.7 `uint8_t mmc_csd_t::maxWriteBlockLength`

write data block length [25:22]

51.0.37 `mmc_extended_csd_config_t` Struct Reference

MMC Extended CSD configuration.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- mmc_command_set_t **commandSet**
Command set.
- uint8_t **ByteValue**
The value to set.
- uint8_t **ByteIndex**
The byte index in Extended CSD(mmc_extended_csd_index_t)
- mmc_extended_csd_access_mode_t **accessMode**
Access mode.

51.0.37.1 Detailed Description

51.0.38 mmc_extended_csd_t Struct Reference

MMC card Extended CSD register (unit: byte).

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- uint8_t **partitionAttribute**
< secure removal type[16]
- uint8_t **userWP**
< max enhance area size [159-157]
- uint8_t **bootPartitionWP**
boot write protect register[173]
- uint8_t **bootWPStatus**
boot write protect status register[174]
- uint8_t **highDensityEraseGroupDefinition**
High-density erase group definition [175].
- uint8_t **bootDataBusConditions**
Boot bus conditions [177].
- uint8_t **bootConfigProtect**
Boot config protection [178].
- uint8_t **partitionConfig**
Boot configuration [179].
- uint8_t **eraseMemoryContent**
Erasered memory content [181].
- uint8_t **dataBusWidth**
Data bus width mode [183].
- uint8_t **highSpeedTiming**
High-speed interface timing [185].
- uint8_t **powerClass**
Power class [187].
- uint8_t **commandSetRevision**

- **uint8_t commandSet**
Command set revision [189].
- **uint8_t extendecCsdVersion**
Extended CSD revision [192].
- **uint8_t csdStructureVersion**
CSD structure version [194].
- **uint8_t cardType**
Card Type [196].
- **uint8_t ioDriverStrength**
IO driver strength [197].
- **uint8_t powerClass52MHz195V**
< out of interrupt busy timing [198]
- **uint8_t powerClass26MHz195V**
Power Class for 26MHz @ 1.95V [201].
- **uint8_t powerClass52MHz360V**
Power Class for 52MHz @ 3.6V [202].
- **uint8_t powerClass26MHz360V**
Power Class for 26MHz @ 3.6V [203].
- **uint8_t minimumReadPerformance4Bit26MHz**
Minimum Read Performance for 4bit at 26MHz [205].
- **uint8_t minimumWritePerformance4Bit26MHz**
Minimum Write Performance for 4bit at 26MHz [206].
- **uint8_t minimumReadPerformance8Bit26MHz4Bit52MHz**
Minimum read Performance for 8bit at 26MHz/4bit @ 52MHz [207].
- **uint8_t minimumWritePerformance8Bit26MHz4Bit52MHz**
Minimum Write Performance for 8bit at 26MHz/4bit @ 52MHz [208].
- **uint8_t minimumReadPerformance8Bit52MHz**
Minimum Read Performance for 8bit at 52MHz [209].
- **uint8_t minimumWritePerformance8Bit52MHz**
Minimum Write Performance for 8bit at 52MHz [210].
- **uint32_t sectorCount**
Sector Count [215:212].
- **uint8_t sleepAwakeTimeout**
< sleep notification timeout [216]
- **uint8_t sleepCurrentVCCQ**
< Production state awareness timeout [218]
- **uint8_t sleepCurrentVCC**
Sleep current (VCC) [220].
- **uint8_t highCapacityWriteProtectGroupSize**
High-capacity write protect group size [221].
- **uint8_t reliableWriteSectorCount**
Reliable write sector count [222].
- **uint8_t highCapacityEraseTimeout**
High-capacity erase timeout [223].
- **uint8_t highCapacityEraseUnitSize**
High-capacity erase unit size [224].
- **uint8_t accessSize**
Access size [225].
- **uint8_t minReadPerformance8bitAt52MHZDDR**
< secure trim multiplier[229]

- `uint8_t minWritePerformance8bitAt52MHZDDR`
Minimum write performance for 8bit at DDR 52MHZ[235].
- `uint8_t powerClass200MHZVCCQ130VVCC360V`
power class for 200MHZ, at VCCQ= 1.3V,VCC=3.6V[236]
- `uint8_t powerClass200MHZVCCQ195VVCC360V`
power class for 200MHZ, at VCCQ= 1.95V,VCC=3.6V[237]
- `uint8_t powerClass52MHZDDR195V`
power class for 52MHZ,DDR at Vcc 1.95V[238]
- `uint8_t powerClass52MHZDDR360V`
power class for 52MHZ,DDR at Vcc 3.6V[239]
- `uint32_t cacheSize`
< 1st initialization time after partitioning[241]
- `uint8_t powerClass200MHZDDR360V`
power class for 200MHZ, DDR at VCC=2.6V[253]
- `uint8_t extPartitionSupport`
< fw VERSION [261-254]
- `uint8_t supportedCommandSet`
< large unit size[495]

51.0.38.1 Detailed Description

51.0.38.2 Field Documentation

51.0.38.2.1 `uint8_t mmc_extended_csd_t::partitionAttribute`

< product state awareness enablement[17]

< max preload data size[21-18]

< pre-load data size[25-22]

< FFU status [26]

< mode operation code[29]

< mode config [30]

< control to turn on/off cache[33]

< power off notification[34]

< packed cmd fail index [35]

< packed cmd status[36]

< context configuration[51-37]

< extended partitions attribut[53-52]

< exception events status[55-54]

< exception events control[57-56]

< number of group to be released[58]

< class 6 command control[59]
< 1st initialization after disabling sector size emu[60]
< sector size[61]
< sector size emulation[62]
< native sector size[63]
< period wakeup [131]
< package case temperature is controlled[132]
< production state awareness[133]
< enhanced user data start addr [139-136]
< enhanced user data area size[142-140]
< general purpose partition size[154-153] partition attribute [156]

51.0.38.2.2 uint8_t mmc_extended_csd_t::userWP

< HPI management [161]
< write reliability parameter register[166]
< write reliability setting register[167]
< RPMB size multi [168]
< FW configuration[169] user write protect register[171]

51.0.38.2.3 uint8_t mmc_extended_csd_t::powerClass52MHz195V

< partition switch timing [199] Power Class for 52MHz @ 1.95V [200]

51.0.38.2.4 uint8_t mmc_extended_csd_t::sleepAwakeTimeout

Sleep/awake timeout [217]

51.0.38.2.5 uint8_t mmc_extended_csd_t::sleepCurrentVCCQ

Sleep current (VCCQ) [219]

51.0.38.2.6 uint8_t mmc_extended_csd_t::minReadPerformance8bitAt52MHZDDR

< secure erase multiplier[230]
< secure feature support[231]

< trim multiplier[232] Minimum read performance for 8bit at DDR 52MHZ[234]

51.0.38.2.7 uint32_t mmc_extended_csd_t::cacheSize

< correct prg sectors number[245-242]

< background operations status[246]

< power off notification timeout[247]

< generic CMD6 timeout[248] cache size[252-249]

51.0.38.2.8 uint8_t mmc_extended_csd_t::extPartitionSupport

< device version[263-262]

< optimal trim size[264]

< optimal write size[265]

< optimal read size[266]

< pre EOL information[267]

< device life time estimation typeA[268]

< device life time estimation typeB[269]

< number of FW sectors correctly programmed[305-302]

< FFU argument[490-487]

< operation code timeout[491]

< support mode [493] extended partition attribute support[494]

51.0.38.2.9 uint8_t mmc_extended_csd_t::supportedCommandSet

< context management capability[496]

< tag resource size[497]

< tag unit size[498]

< max packed write cmd[500]

< max packed read cmd[501]

< HPI feature[503] Supported Command Sets [504]

51.0.39 sd_cid_t Struct Reference

SD card CID register.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- `uint8_t manufacturerID`
Manufacturer ID [127:120].
- `uint16_t applicationID`
OEM/Application ID [119:104].
- `uint8_t productName [SD_PRODUCT_NAME_BYTES]`
Product name [103:64].
- `uint8_t productVersion`
Product revision [63:56].
- `uint32_t productSerialNumber`
Product serial number [55:24].
- `uint16_t manufacturerData`
Manufacturing date [19:8].

51.0.40 `sd_csd_t` Struct Reference

SD card CSD register.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- `uint8_t csdStructure`
CSD structure [127:126].
- `uint8_t dataReadAccessTime1`
Data read access-time-1 [119:112].
- `uint8_t dataReadAccessTime2`
*Data read access-time-2 in clock cycles (NSAC*100) [111:104].*
- `uint8_t transferSpeed`
Maximum data transfer rate [103:96].
- `uint16_t cardCommandClass`
Card command classes [95:84].
- `uint8_t readBlockLength`
Maximum read data block length [83:80].
- `uint16_t flags`
Flags in _sd_csd_flag.
- `uint32_t deviceSize`
Device size [73:62].
- `uint8_t readCurrentVddMin`
Maximum read current at VDD min [61:59].
- `uint8_t readCurrentVddMax`
Maximum read current at VDD max [58:56].
- `uint8_t writeCurrentVddMin`
Maximum write current at VDD min [55:53].
- `uint8_t writeCurrentVddMax`

- `uint8_t deviceSizeMultiplier`
Maximum write current at VDD max [52:50].
- `uint8_t eraseSectorSize`
Erase sector size [49:47].
- `uint8_t writeProtectGroupSize`
Write protect group size [45:39].
- `uint8_t writeSpeedFactor`
Write speed factor [38:32].
- `uint8_t writeBlockLength`
Maximum write data block length [28:26].
- `uint8_t fileFormat`
File format [25:22].

File format [11:10].

51.0.41 `sd_scr_t` Struct Reference

SD card SCR register.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- `uint8_t scrStructure`
SCR Structure [63:60].
- `uint8_t sdSpecification`
SD memory card specification version [59:56].
- `uint16_t flags`
SCR flags in _sd_scr_flag.
- `uint8_t sdSecurity`
Security specification supported [54:52].
- `uint8_t sdBusWidths`
Data bus widths supported [51:48].
- `uint8_t extendedSecurity`
Extended security support [46:43].
- `uint8_t commandSupport`
Command support bits [33:32] 33-support CMD23, 32-support cmd20.
- `uint32_t reservedForManufacturer`
reserved for manufacturer usage [31:0]

51.0.42 `sd_status_t` Struct Reference

SD card status.

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- `uint8_t busWidth`
current buswidth
- `uint8_t secureMode`
secured mode
- `uint16_t cardType`
sdcard type
- `uint32_t protectedSize`
size of protected area
- `uint8_t speedClass`
speed class of card
- `uint8_t performanceMove`
Performance of move indicated by 1[MB/S]step.
- `uint8_t auSize`
size of AU
- `uint16_t eraseSize`
number of AUs to be erased at a time
- `uint8_t eraseTimeout`
timeout value for erasing areas specified by UNIT OF ERASE AU
- `uint8_t eraseOffset`
fixed offset value added to erase time
- `uint8_t uhsSpeedGrade`
speed grade for UHS mode
- `uint8_t uhsAuSize`
size of AU for UHS mode

51.0.43 `sdio_common_cis_t` Struct Reference

sdio card common CIS

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- `uint16_t mID`
manufacturer code
- `uint16_t mInfo`
manufacturer information
- `uint8_t funcID`
function ID
- `uint16_t fn0MaxBlkSize`
function 0 max block size
- `uint8_t maxTransSpeed`
max data transfer speed for all function

51.0.44 sdio_fbr_t Struct Reference

sdio card FBR register

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- uint8_t **flags**
current io flags
- uint8_t **ioStdFunctionCode**
current io standard function code
- uint8_t **ioExtFunctionCode**
current io extended function code
- uint32_t **ioPointerToCIS**
current io pointer to CIS
- uint32_t **ioPointerToCSA**
current io pointer to CSA
- uint16_t **ioBlockSize**
current io block size

51.0.45 sdio_func_cis_t Struct Reference

sdio card function CIS

```
#include <fsl_sdmmc_spec.h>
```

Data Fields

- uint8_t **funcID**
function ID
- uint8_t **funcInfo**
function info
- uint8_t **ioVersion**
level of application specification this io support
- uint32_t **cardPSN**
product serial number
- uint32_t **ioCSASize**
available CSA size for io
- uint8_t **ioCSAProperty**
CSA property.
- uint16_t **ioMaxBlockSize**
io max transfer data size
- uint32_t **ioOCR**
io ioeration condition
- uint8_t **ioOPMinPwr**
min current in operation mode
- uint8_t **ioOPAvgPwr**

- **uint8_t ioOPMaxPwr**
average current in operation mode
- **uint8_t ioSBMinPwr**
max current in operation mode
- **uint8_t ioSBAvgPwr**
min current in standby mode
- **uint8_t ioSBMaxPwr**
average current in standby mode
- **uint16_t ioMinBandWidth**
max current in standby mode
- **uint16_t ioOptimumBandWidth**
io min transfer bandwidth
- **uint16_t ioReadyTimeout**
io optimum transfer bandwidth
- **uint16_t ioHighCurrentAvgCurrent**
timeout value from enable to ready
the average peak current (mA)
when IO operating in high current mode
- **uint16_t ioHighCurrentMaxCurrent**
the max peak current (mA)
when IO operating in high current mode
- **uint16_t ioLowCurrentAvgCurrent**
the average peak current (mA)
when IO operating in lower current mode
- **uint16_t ioLowCurrentMaxCurrent**
the max peak current (mA)
when IO operating in lower current mode

How to Reach Us:

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, Freescale, the Freescale logo, Kinetis, Processor Expert, and Tower are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, Cortex, Keil, Mbed, Mbed Enabled, and Vision are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2018 NXP B.V.