

---

# MCUXpresso SDK API Reference Manual

**NXP Semiconductors**

Document Number: MCUXSDKLPC51U68APIRM

Rev. 0

Feb 2018





# Contents

Chapter [Introduction](#)

Chapter [Driver errors status](#)

Chapter [Trademarks](#)

Chapter [Architectural Overview](#)

Chapter [FLEXCOMM: FLEXCOMM Driver](#)

[5.1 Overview](#) . . . . . **11**

[5.2 FLEXCOMM Driver](#) . . . . . **12**

Chapter [I2C: Inter-Integrated Circuit Driver](#)

[6.1 Overview](#) . . . . . **13**

[6.2 Typical use case](#) . . . . . **13**

[6.2.1 Master Operation in functional method](#) . . . . . 13

[6.2.2 Master Operation in interrupt transactional method](#) . . . . . 14

[6.2.3 Master Operation in DMA transactional method](#) . . . . . 15

[6.2.4 Slave Operation in functional method](#) . . . . . 15

[6.2.5 Slave Operation in interrupt transactional method](#) . . . . . 16

[6.3 I2C Driver](#) . . . . . **18**

[6.3.1 Overview](#) . . . . . 18

[6.3.2 Macro Definition Documentation](#) . . . . . 19

[6.3.3 Enumeration Type Documentation](#) . . . . . 19

[6.4 I2C Master Driver](#) . . . . . **20**

[6.4.1 Overview](#) . . . . . 20

[6.4.2 Data Structure Documentation](#) . . . . . 22

[6.4.3 Typedef Documentation](#) . . . . . 25

[6.4.4 Enumeration Type Documentation](#) . . . . . 26

[6.4.5 Function Documentation](#) . . . . . 27

[6.5 I2C Slave Driver](#) . . . . . **36**

# Contents

Section Number	Title	Page Number
6.5.1	Overview . . . . .	36
6.5.2	Data Structure Documentation . . . . .	38
6.5.3	Typedef Documentation . . . . .	41
6.5.4	Enumeration Type Documentation . . . . .	41
6.5.5	Function Documentation . . . . .	43
<b>6.6</b>	<b>I2C DMA Driver . . . . .</b>	<b>51</b>
6.6.1	Overview . . . . .	51
6.6.2	Data Structure Documentation . . . . .	51
6.6.3	Typedef Documentation . . . . .	52
6.6.4	Function Documentation . . . . .	52
<b>6.7</b>	<b>I2C FreeRTOS Driver . . . . .</b>	<b>55</b>
6.7.1	Overview . . . . .	55
6.7.2	Data Structure Documentation . . . . .	55
6.7.3	Function Documentation . . . . .	55
<b>Chapter I2S: I2S Driver</b>		
<b>7.1</b>	<b>Overview . . . . .</b>	<b>57</b>
<b>7.2</b>	<b>I2S Driver Initialization and Configuration . . . . .</b>	<b>57</b>
<b>7.3</b>	<b>I2S Transmit Data . . . . .</b>	<b>57</b>
<b>7.4</b>	<b>I2S Interrupt related functions . . . . .</b>	<b>58</b>
<b>7.5</b>	<b>I2S Other functions . . . . .</b>	<b>58</b>
<b>7.6</b>	<b>I2S Data formats . . . . .</b>	<b>58</b>
7.6.1	DMA mode . . . . .	58
7.6.2	Interrupt mode . . . . .	60
<b>7.7</b>	<b>I2S Driver Examples . . . . .</b>	<b>61</b>
7.7.1	Interrupt mode examples . . . . .	61
7.7.2	DMA mode examples . . . . .	62
<b>7.8</b>	<b>I2S Driver . . . . .</b>	<b>64</b>
7.8.1	Overview . . . . .	64
7.8.2	Data Structure Documentation . . . . .	66
7.8.3	Macro Definition Documentation . . . . .	68
7.8.4	Typedef Documentation . . . . .	68
7.8.5	Enumeration Type Documentation . . . . .	69
7.8.6	Function Documentation . . . . .	70
<b>7.9</b>	<b>I2S DMA Driver . . . . .</b>	<b>77</b>

# Contents

Section Number	Title	Page Number
7.9.1	Overview . . . . .	77
7.9.2	Data Structure Documentation . . . . .	77
7.9.3	Typedef Documentation . . . . .	78
7.9.4	Function Documentation . . . . .	78

## Chapter SPI: Serial Peripheral Interface Driver

<b>8.1</b>	<b>Overview . . . . .</b>	<b>81</b>
<b>8.2</b>	<b>Typical use case . . . . .</b>	<b>81</b>
8.2.1	SPI master transfer using an interrupt method . . . . .	81
8.2.2	SPI Send/receive using a DMA method . . . . .	82
<b>8.3</b>	<b>SPI Driver . . . . .</b>	<b>84</b>
8.3.1	Overview . . . . .	84
8.3.2	Data Structure Documentation . . . . .	89
8.3.3	Macro Definition Documentation . . . . .	92
8.3.4	Enumeration Type Documentation . . . . .	92
8.3.5	Function Documentation . . . . .	95
<b>8.4</b>	<b>SPI DMA Driver . . . . .</b>	<b>105</b>
8.4.1	Overview . . . . .	105
8.4.2	Data Structure Documentation . . . . .	106
8.4.3	Typedef Documentation . . . . .	106
8.4.4	Function Documentation . . . . .	106
<b>8.5</b>	<b>SPI FreeRTOS driver . . . . .</b>	<b>111</b>
8.5.1	Overview . . . . .	111
8.5.2	Data Structure Documentation . . . . .	111
8.5.3	Function Documentation . . . . .	112

## Chapter USART: Universal Asynchronous Receiver/Transmitter Driver

<b>9.1</b>	<b>Overview . . . . .</b>	<b>115</b>
<b>9.2</b>	<b>Typical use case . . . . .</b>	<b>116</b>
9.2.1	USART Send/receive using a polling method . . . . .	116
9.2.2	USART Send/receive using an interrupt method . . . . .	116
9.2.3	USART Receive using the ringbuffer feature . . . . .	117
9.2.4	USART Send/Receive using the DMA method . . . . .	118
<b>9.3</b>	<b>USART Driver . . . . .</b>	<b>120</b>
9.3.1	Overview . . . . .	120
9.3.2	Data Structure Documentation . . . . .	123
9.3.3	Macro Definition Documentation . . . . .	126
9.3.4	Typedef Documentation . . . . .	126

# Contents

Section Number	Title	Page Number
9.3.5	Enumeration Type Documentation . . . . .	126
9.3.6	Function Documentation . . . . .	128
<b>9.4</b>	<b>USART DMA Driver . . . . .</b>	<b>139</b>
9.4.1	Overview . . . . .	139
9.4.2	Data Structure Documentation . . . . .	140
9.4.3	Typedef Documentation . . . . .	141
9.4.4	Function Documentation . . . . .	141
<b>9.5</b>	<b>USART FreeRTOS Driver . . . . .</b>	<b>145</b>
9.5.1	Overview . . . . .	145
9.5.2	Data Structure Documentation . . . . .	145
9.5.3	Function Documentation . . . . .	146
<b>Chapter</b>	<b>DMA: Direct Memory Access Controller Driver</b>	
<b>10.1</b>	<b>Overview . . . . .</b>	<b>149</b>
<b>10.2</b>	<b>Typical use case . . . . .</b>	<b>149</b>
10.2.1	DMA Operation . . . . .	149
<b>10.3</b>	<b>Data Structure Documentation . . . . .</b>	<b>152</b>
10.3.1	struct dma_descriptor_t . . . . .	152
10.3.2	struct dma_xfercfg_t . . . . .	152
10.3.3	struct dma_channel_trigger_t . . . . .	153
10.3.4	struct dma_transfer_config_t . . . . .	153
10.3.5	struct dma_handle_t . . . . .	154
<b>10.4</b>	<b>Macro Definition Documentation . . . . .</b>	<b>154</b>
10.4.1	FSL_DMA_DRIVER_VERSION . . . . .	154
<b>10.5</b>	<b>Typedef Documentation . . . . .</b>	<b>154</b>
10.5.1	dma_callback . . . . .	154
<b>10.6</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>154</b>
10.6.1	dma_priority_t . . . . .	154
10.6.2	dma_irq_t . . . . .	155
10.6.3	dma_trigger_type_t . . . . .	155
10.6.4	dma_trigger_burst_t . . . . .	155
10.6.5	dma_burst_wrap_t . . . . .	155
10.6.6	dma_transfer_type_t . . . . .	156
10.6.7	_dma_transfer_status . . . . .	156
<b>10.7</b>	<b>Function Documentation . . . . .</b>	<b>156</b>
10.7.1	DMA_Init . . . . .	156
10.7.2	DMA_Deinit . . . . .	156

# Contents

Section Number	Title	Page Number
10.7.3	DMA_ChannelIsActive . . . . .	156
10.7.4	DMA_EnableChannelInterrupts . . . . .	157
10.7.5	DMA_DisableChannelInterrupts . . . . .	157
10.7.6	DMA_EnableChannel . . . . .	157
10.7.7	DMA_DisableChannel . . . . .	157
10.7.8	DMA_EnableChannelPeriphRq . . . . .	158
10.7.9	DMA_DisableChannelPeriphRq . . . . .	158
10.7.10	DMA_ConfigureChannelTrigger . . . . .	158
10.7.11	DMA_GetRemainingBytes . . . . .	159
10.7.12	DMA_SetChannelPriority . . . . .	159
10.7.13	DMA_GetChannelPriority . . . . .	159
10.7.14	DMA_CreateDescriptor . . . . .	159
10.7.15	DMA_AbortTransfer . . . . .	160
10.7.16	DMA_CreateHandle . . . . .	160
10.7.17	DMA_SetCallback . . . . .	160
10.7.18	DMA_PrepareTransfer . . . . .	161
10.7.19	DMA_SubmitTransfer . . . . .	161
10.7.20	DMA_StartTransfer . . . . .	162
10.7.21	DMA_HandleIRQ . . . . .	162
<b>Chapter</b>	<b>SCTimer: SCTimer/PWM (SCT)</b>	
<b>11.1</b>	<b>Overview . . . . .</b>	<b>163</b>
<b>11.2</b>	<b>Function groups . . . . .</b>	<b>163</b>
11.2.1	Initialization and deinitialization . . . . .	163
11.2.2	PWM Operations . . . . .	163
11.2.3	Status . . . . .	163
11.2.4	Interrupt . . . . .	163
<b>11.3</b>	<b>SCTimer State machine and operations . . . . .</b>	<b>164</b>
11.3.1	SCTimer event operations . . . . .	164
11.3.2	SCTimer state operations . . . . .	164
11.3.3	SCTimer action operations . . . . .	164
<b>11.4</b>	<b>16-bit counter mode . . . . .</b>	<b>164</b>
<b>11.5</b>	<b>Typical use case . . . . .</b>	<b>165</b>
11.5.1	PWM output . . . . .	165
<b>11.6</b>	<b>Data Structure Documentation . . . . .</b>	<b>169</b>
11.6.1	struct sctimer_pwm_signal_param_t . . . . .	169
11.6.2	struct sctimer_config_t . . . . .	170
<b>11.7</b>	<b>Typedef Documentation . . . . .</b>	<b>170</b>
11.7.1	sctimer_event_callback_t . . . . .	170

# Contents

Section Number	Title	Page Number
<b>11.8</b>	<b>Enumeration Type Documentation</b>	<b>171</b>
11.8.1	sctimer_pwm_mode_t	171
11.8.2	sctimer_counter_t	171
11.8.3	sctimer_input_t	171
11.8.4	sctimer_out_t	171
11.8.5	sctimer_pwm_level_select_t	172
11.8.6	sctimer_clock_mode_t	172
11.8.7	sctimer_clock_select_t	172
11.8.8	sctimer_conflict_resolution_t	172
11.8.9	sctimer_interrupt_enable_t	173
11.8.10	sctimer_status_flags_t	173
<b>11.9</b>	<b>Function Documentation</b>	<b>174</b>
11.9.1	SCTIMER_Init	174
11.9.2	SCTIMER_Deinit	174
11.9.3	SCTIMER_GetDefaultConfig	174
11.9.4	SCTIMER_SetupPwm	175
11.9.5	SCTIMER_UpdatePwmDutycycle	175
11.9.6	SCTIMER_EnableInterrupts	176
11.9.7	SCTIMER_DisableInterrupts	176
11.9.8	SCTIMER_GetEnabledInterrupts	176
11.9.9	SCTIMER_GetStatusFlags	177
11.9.10	SCTIMER_ClearStatusFlags	177
11.9.11	SCTIMER_StartTimer	177
11.9.12	SCTIMER_StopTimer	178
11.9.13	SCTIMER_CreateAndScheduleEvent	178
11.9.14	SCTIMER_ScheduleEvent	179
11.9.15	SCTIMER_IncreaseState	179
11.9.16	SCTIMER_GetCurrentState	179
11.9.17	SCTIMER_SetupCaptureAction	180
11.9.18	SCTIMER_SetCallback	180
11.9.19	SCTIMER_SetupNextStateAction	181
11.9.20	SCTIMER_SetupOutputSetAction	181
11.9.21	SCTIMER_SetupOutputClearAction	181
11.9.22	SCTIMER_SetupOutputToggleAction	182
11.9.23	SCTIMER_SetupCounterLimitAction	182
11.9.24	SCTIMER_SetupCounterStopAction	182
11.9.25	SCTIMER_SetupCounterStartAction	183
11.9.26	SCTIMER_SetupCounterHaltAction	183
11.9.27	SCTIMER_SetupDmaTriggerAction	183
11.9.28	SCTIMER_EventHandleIRQ	184
<b>Chapter</b>	<b>WWDT: Windowed Watchdog Timer Driver</b>	
<b>12.1</b>	<b>Overview</b>	<b>185</b>



# Contents

Section Number	Title	Page Number
<b>12.2</b>	<b>Function groups</b>	<b>185</b>
12.2.1	Initialization and deinitialization	185
12.2.2	Status	185
12.2.3	Interrupt	185
12.2.4	Watch dog Refresh	185
<b>12.3</b>	<b>Typical use case</b>	<b>185</b>
<b>12.4</b>	<b>Data Structure Documentation</b>	<b>187</b>
12.4.1	struct wwdt_config_t	187
<b>12.5</b>	<b>Macro Definition Documentation</b>	<b>187</b>
12.5.1	FSL_WWDT_DRIVER_VERSION	187
<b>12.6</b>	<b>Enumeration Type Documentation</b>	<b>187</b>
12.6.1	_wwdt_status_flags_t	187
<b>12.7</b>	<b>Function Documentation</b>	<b>187</b>
12.7.1	WWDT_GetDefaultConfig	187
12.7.2	WWDT_Init	188
12.7.3	WWDT_Deinit	188
12.7.4	WWDT_Enable	189
12.7.5	WWDT_Disable	189
12.7.6	WWDT_GetStatusFlags	189
12.7.7	WWDT_ClearStatusFlags	190
12.7.8	WWDT_SetWarningValue	190
12.7.9	WWDT_SetTimeoutValue	190
12.7.10	WWDT_SetWindowValue	191
12.7.11	WWDT_Refresh	191
<b>Chapter</b>	<b>RTC: Real Time Clock</b>	
<b>13.1</b>	<b>Overview</b>	<b>193</b>
<b>13.2</b>	<b>Function groups</b>	<b>193</b>
13.2.1	Initialization and deinitialization	193
13.2.2	Set & Get Datetime	193
13.2.3	Set & Get Alarm	193
13.2.4	Start & Stop timer	193
13.2.5	Status	194
13.2.6	Interrupt	194
13.2.7	High resolution timer	194
<b>13.3</b>	<b>Typical use case</b>	<b>194</b>
13.3.1	RTC tick example	194

# Contents

Section Number	Title	Page Number
<b>13.4</b>	<b>Data Structure Documentation</b>	<b>196</b>
13.4.1	struct rtc_datetime_t	196
<b>13.5</b>	<b>Enumeration Type Documentation</b>	<b>196</b>
13.5.1	rtc_interrupt_enable_t	196
13.5.2	rtc_status_flags_t	196
<b>13.6</b>	<b>Function Documentation</b>	<b>197</b>
13.6.1	RTC_Init	197
13.6.2	RTC_Deinit	197
13.6.3	RTC_SetDatetime	197
13.6.4	RTC_GetDatetime	197
13.6.5	RTC_SetAlarm	198
13.6.6	RTC_GetAlarm	198
13.6.7	RTC_SetWakeupCount	198
13.6.8	RTC_GetWakeupCount	199
13.6.9	RTC_EnableInterrupts	199
13.6.10	RTC_DisableInterrupts	199
13.6.11	RTC_GetEnabledInterrupts	199
13.6.12	RTC_GetStatusFlags	200
13.6.13	RTC_ClearStatusFlags	200
13.6.14	RTC_StartTimer	200
13.6.15	RTC_StopTimer	201
13.6.16	RTC_Reset	201
<b>Chapter</b>	<b>MRT: Multi-Rate Timer</b>	
<b>14.1</b>	<b>Overview</b>	<b>203</b>
<b>14.2</b>	<b>Function groups</b>	<b>203</b>
14.2.1	Initialization and deinitialization	203
14.2.2	Timer period Operations	203
14.2.3	Start and Stop timer operations	203
14.2.4	Get and release channel	204
14.2.5	Status	204
14.2.6	Interrupt	204
<b>14.3</b>	<b>Typical use case</b>	<b>204</b>
14.3.1	MRT tick example	204
<b>14.4</b>	<b>Data Structure Documentation</b>	<b>206</b>
14.4.1	struct mrt_config_t	206
<b>14.5</b>	<b>Enumeration Type Documentation</b>	<b>206</b>
14.5.1	mrt_chnl_t	206
14.5.2	mrt_timer_mode_t	206

# Contents

Section Number	Title	Page Number
14.5.3	mrt_interrupt_enable_t . . . . .	207
14.5.4	mrt_status_flags_t . . . . .	207
<b>14.6</b>	<b>Function Documentation . . . . .</b>	<b>207</b>
14.6.1	MRT_Init . . . . .	207
14.6.2	MRT_Deinit . . . . .	207
14.6.3	MRT_GetDefaultConfig . . . . .	207
14.6.4	MRT_SetupChannelMode . . . . .	208
14.6.5	MRT_EnableInterrupts . . . . .	208
14.6.6	MRT_DisableInterrupts . . . . .	208
14.6.7	MRT_GetEnabledInterrupts . . . . .	209
14.6.8	MRT_GetStatusFlags . . . . .	210
14.6.9	MRT_ClearStatusFlags . . . . .	210
14.6.10	MRT_UpdateTimerPeriod . . . . .	210
14.6.11	MRT_GetCurrentTimerCount . . . . .	211
14.6.12	MRT_StartTimer . . . . .	211
14.6.13	MRT_StopTimer . . . . .	212
14.6.14	MRT_GetIdleChannel . . . . .	212
14.6.15	MRT_ReleaseChannel . . . . .	212
<b>Chapter</b>	<b>ADC: 12-bit SAR Analog-to-Digital Converter Driver</b>	
<b>15.1</b>	<b>Overview . . . . .</b>	<b>215</b>
<b>15.2</b>	<b>Typical use case . . . . .</b>	<b>215</b>
15.2.1	Polling Configuration . . . . .	215
15.2.2	Interrupt Configuration . . . . .	215
<b>15.3</b>	<b>Data Structure Documentation . . . . .</b>	<b>219</b>
15.3.1	struct adc_config_t . . . . .	219
15.3.2	struct adc_conv_seq_config_t . . . . .	219
15.3.3	struct adc_result_info_t . . . . .	220
<b>15.4</b>	<b>Macro Definition Documentation . . . . .</b>	<b>220</b>
15.4.1	LPC_ADC_DRIVER_VERSION . . . . .	220
<b>15.5</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>221</b>
15.5.1	_adc_status_flags . . . . .	221
15.5.2	_adc_interrupt_enable . . . . .	222
15.5.3	adc_trigger_polarity_t . . . . .	222
15.5.4	adc_priority_t . . . . .	222
15.5.5	adc_seq_interrupt_mode_t . . . . .	222
15.5.6	adc_threshold_compare_status_t . . . . .	223
15.5.7	adc_threshold_crossing_status_t . . . . .	223
15.5.8	adc_threshold_interrupt_mode_t . . . . .	223

# Contents

Section Number	Title	Page Number
<b>15.6</b>	<b>Function Documentation</b>	<b>223</b>
15.6.1	ADC_Init	223
15.6.2	ADC_Deinit	224
15.6.3	ADC_GetDefaultConfig	224
15.6.4	ADC_DoSelfCalibration	224
15.6.5	ADC_EnableTemperatureSensor	225
15.6.6	ADC_EnableConvSeqA	225
15.6.7	ADC_SetConvSeqAConfig	225
15.6.8	ADC_DoSoftwareTriggerConvSeqA	226
15.6.9	ADC_EnableConvSeqABurstMode	226
15.6.10	ADC_SetConvSeqAHighPriority	226
15.6.11	ADC_EnableConvSeqB	226
15.6.12	ADC_SetConvSeqBConfig	227
15.6.13	ADC_DoSoftwareTriggerConvSeqB	227
15.6.14	ADC_EnableConvSeqBBurstMode	227
15.6.15	ADC_SetConvSeqBHighPriority	227
15.6.16	ADC_GetConvSeqAGlobalConversionResult	228
15.6.17	ADC_GetConvSeqBGlobalConversionResult	228
15.6.18	ADC_GetChannelConversionResult	228
15.6.19	ADC_SetThresholdPair0	229
15.6.20	ADC_SetThresholdPair1	229
15.6.21	ADC_SetChannelWithThresholdPair0	229
15.6.22	ADC_SetChannelWithThresholdPair1	230
15.6.23	ADC_EnableInterrupts	230
15.6.24	ADC_DisableInterrupts	230
15.6.25	ADC_EnableShresholdCompareInterrupt	231
15.6.26	ADC_EnableThresholdCompareInterrupt	231
15.6.27	ADC_GetStatusFlags	232
15.6.28	ADC_ClearStatusFlags	232
<b>Chapter</b>	<b>FLASHIAP: Flash In Application Programming Driver</b>	
<b>16.1</b>	<b>Overview</b>	<b>233</b>
<b>16.2</b>	<b>GFlash In Application Programming operation</b>	<b>233</b>
<b>16.3</b>	<b>Typical use case</b>	<b>233</b>
<b>16.4</b>	<b>Macro Definition Documentation</b>	<b>235</b>
16.4.1	FSL_FLASHIAP_DRIVER_VERSION	235
<b>16.5</b>	<b>Enumeration Type Documentation</b>	<b>235</b>
16.5.1	_flashiap_status	235
16.5.2	_flashiap_commands	236

# Contents

Section Number	Title	Page Number
<b>16.6</b>	<b>Function Documentation</b>	<b>236</b>
16.6.1	iap_entry	236
16.6.2	FLASHIAP_PrepareSectorForWrite	236
16.6.3	FLASHIAP_CopyRamToFlash	237
16.6.4	FLASHIAP_EraseSector	238
16.6.5	FLASHIAP_ErasePage	239
16.6.6	FLASHIAP_BlankCheckSector	240
16.6.7	FLASHIAP_Compare	240
<b>Chapter</b>	<b>UTICK: MicroTick Timer Driver</b>	
<b>17.1</b>	<b>Overview</b>	<b>243</b>
<b>17.2</b>	<b>Typical use case</b>	<b>243</b>
<b>17.3</b>	<b>Macro Definition Documentation</b>	<b>244</b>
17.3.1	FSL_UTICK_DRIVER_VERSION	244
<b>17.4</b>	<b>Typedef Documentation</b>	<b>244</b>
17.4.1	utick_callback_t	244
<b>17.5</b>	<b>Enumeration Type Documentation</b>	<b>244</b>
17.5.1	utick_mode_t	244
<b>17.6</b>	<b>Function Documentation</b>	<b>244</b>
17.6.1	UTICK_Init	244
17.6.2	UTICK_Deinit	244
17.6.3	UTICK_GetStatusFlags	244
17.6.4	UTICK_ClearStatusFlags	245
17.6.5	UTICK_SetTick	245
17.6.6	UTICK_HandleIRQ	245
<b>Chapter</b>	<b>FMEAS: Frequency Measure Driver</b>	
<b>18.1</b>	<b>Overview</b>	<b>247</b>
<b>18.2</b>	<b>Frequency Measure Driver operation</b>	<b>247</b>
<b>18.3</b>	<b>Typical use case</b>	<b>247</b>
<b>18.4</b>	<b>Macro Definition Documentation</b>	<b>248</b>
18.4.1	FSL_FMEAS_DRIVER_VERSION	248
<b>18.5</b>	<b>Function Documentation</b>	<b>248</b>
18.5.1	FMEAS_StartMeasure	248
18.5.2	FMEAS_IsMeasureComplete	248

# Contents

Section Number	Title	Page Number
18.5.3	FMEAS_GetFrequency . . . . .	248
<b>Chapter</b>	<b>CRC: Cyclic Redundancy Check Driver</b>	
<b>19.1</b>	<b>Overview . . . . .</b>	<b>251</b>
<b>19.2</b>	<b>CRC Driver Initialization and Configuration . . . . .</b>	<b>251</b>
<b>19.3</b>	<b>CRC Write Data . . . . .</b>	<b>251</b>
<b>19.4</b>	<b>CRC Get Checksum . . . . .</b>	<b>251</b>
<b>19.5</b>	<b>Comments about API usage in RTOS . . . . .</b>	<b>252</b>
<b>19.6</b>	<b>Data Structure Documentation . . . . .</b>	<b>253</b>
19.6.1	struct crc_config_t . . . . .	253
<b>19.7</b>	<b>Macro Definition Documentation . . . . .</b>	<b>254</b>
19.7.1	FSL_CRC_DRIVER_VERSION . . . . .	254
19.7.2	CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT . . . . .	254
<b>19.8</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>254</b>
19.8.1	crc_polynomial_t . . . . .	254
<b>19.9</b>	<b>Function Documentation . . . . .</b>	<b>254</b>
19.9.1	CRC_Init . . . . .	254
19.9.2	CRC_Deinit . . . . .	255
19.9.3	CRC_Reset . . . . .	255
19.9.4	CRC_GetDefaultConfig . . . . .	255
19.9.5	CRC_GetConfig . . . . .	255
19.9.6	CRC_WriteData . . . . .	256
19.9.7	CRC_Get32bitResult . . . . .	256
19.9.8	CRC_Get16bitResult . . . . .	256
<b>Chapter</b>	<b>INPUTMUX: Input Multiplexing Driver</b>	
<b>20.1</b>	<b>Overview . . . . .</b>	<b>259</b>
<b>20.2</b>	<b>Input Multiplexing Driver operation . . . . .</b>	<b>259</b>
<b>20.3</b>	<b>Typical use case . . . . .</b>	<b>259</b>
<b>20.4</b>	<b>Macro Definition Documentation . . . . .</b>	<b>259</b>
20.4.1	FSL_INPUTMUX_DRIVER_VERSION . . . . .	259
<b>20.5</b>	<b>Function Documentation . . . . .</b>	<b>260</b>
20.5.1	INPUTMUX_Init . . . . .	260

# Contents

Section Number	Title	Page Number
20.5.2	INPUTMUX_AttachSignal . . . . .	261
20.5.3	INPUTMUX_Deinit . . . . .	261
<b>Chapter</b>	<b>IOCON: I/O pin configuration</b>	
<b>21.1</b>	<b>Overview . . . . .</b>	<b>263</b>
<b>21.2</b>	<b>Function groups . . . . .</b>	<b>263</b>
21.2.1	Pin mux set . . . . .	263
21.2.2	Pin mux set . . . . .	263
<b>21.3</b>	<b>Typical use case . . . . .</b>	<b>263</b>
<b>21.4</b>	<b>Data Structure Documentation . . . . .</b>	<b>265</b>
21.4.1	struct iocon_group_t . . . . .	265
<b>21.5</b>	<b>Macro Definition Documentation . . . . .</b>	<b>265</b>
21.5.1	LPC_IOCON_DRIVER_VERSION . . . . .	265
21.5.2	IOCON_FUNC0 . . . . .	265
<b>21.6</b>	<b>Function Documentation . . . . .</b>	<b>265</b>
21.6.1	IOCON_PinMuxSet . . . . .	265
21.6.2	IOCON_SetPinMuxing . . . . .	266
<b>Chapter</b>	<b>SYSCON: System Configuration</b>	
<b>22.1</b>	<b>Overview . . . . .</b>	<b>269</b>
<b>22.2</b>	<b>Clock driver . . . . .</b>	<b>270</b>
22.2.1	Function description . . . . .	270
22.2.2	Typical use case . . . . .	271
<b>22.3</b>	<b>Power driver . . . . .</b>	<b>272</b>
22.3.1	Function description . . . . .	272
22.3.2	Typical use case . . . . .	273
<b>Chapter</b>	<b>GPIO: General Purpose I/O</b>	
<b>23.1</b>	<b>Overview . . . . .</b>	<b>275</b>
<b>23.2</b>	<b>Function groups . . . . .</b>	<b>275</b>
23.2.1	Initialization and deinitialization . . . . .	275
23.2.2	Pin manipulation . . . . .	275
23.2.3	Port manipulation . . . . .	275
23.2.4	Port masking . . . . .	275

# Contents

Section Number	Title	Page Number
<b>23.3</b>	<b>Typical use case</b>	<b>275</b>
<b>23.4</b>	<b>Data Structure Documentation</b>	<b>277</b>
23.4.1	struct gpio_pin_config_t	277
<b>23.5</b>	<b>Macro Definition Documentation</b>	<b>277</b>
23.5.1	FSL_GPIO_DRIVER_VERSION	277
<b>23.6</b>	<b>Enumeration Type Documentation</b>	<b>277</b>
23.6.1	gpio_pin_direction_t	277
<b>23.7</b>	<b>Function Documentation</b>	<b>277</b>
23.7.1	GPIO_PortInit	277
23.7.2	GPIO_Init	278
23.7.3	GPIO_PinInit	278
23.7.4	GPIO_PinWrite	278
23.7.5	GPIO_WritePinOutput	279
23.7.6	GPIO_PinRead	279
23.7.7	GPIO_ReadPinInput	279
23.7.8	GPIO_PortSet	279
23.7.9	GPIO_SetPinsOutput	280
23.7.10	GPIO_PortClear	280
23.7.11	GPIO_ClearPinsOutput	280
23.7.12	GPIO_PortToggle	280
23.7.13	GPIO_TogglePinsOutput	280
<b>Chapter</b>	<b>GINT: Group GPIO Input Interrupt Driver</b>	
<b>24.1</b>	<b>Overview</b>	<b>281</b>
<b>24.2</b>	<b>Group GPIO Input Interrupt Driver operation</b>	<b>281</b>
<b>24.3</b>	<b>Typical use case</b>	<b>281</b>
<b>24.4</b>	<b>Macro Definition Documentation</b>	<b>282</b>
24.4.1	FSL_GINT_DRIVER_VERSION	282
<b>24.5</b>	<b>Typedef Documentation</b>	<b>282</b>
24.5.1	gint_cb_t	282
<b>24.6</b>	<b>Enumeration Type Documentation</b>	<b>282</b>
24.6.1	gint_comb_t	282
24.6.2	gint_trig_t	282
<b>24.7</b>	<b>Function Documentation</b>	<b>283</b>
24.7.1	GINT_Init	283



# Contents

Section Number	Title	Page Number
24.7.2	<a href="#">GINT_SetCtrl</a>	284
24.7.3	<a href="#">GINT_GetCtrl</a>	284
24.7.4	<a href="#">GINT_ConfigPins</a>	285
24.7.5	<a href="#">GINT_GetConfigPins</a>	285
24.7.6	<a href="#">GINT_EnableCallback</a>	286
24.7.7	<a href="#">GINT_DisableCallback</a>	286
24.7.8	<a href="#">GINT_ClrStatus</a>	286
24.7.9	<a href="#">GINT_GetStatus</a>	287
24.7.10	<a href="#">GINT_Deinit</a>	287
<b>Chapter</b>	<b>PINT: Pin Interrupt and Pattern Match Driver</b>	
<b>25.1</b>	<b>Overview</b>	<b>289</b>
<b>25.2</b>	<b>Pin Interrupt and Pattern match Driver operation</b>	<b>289</b>
25.2.1	<a href="#">Pin Interrupt use case</a>	289
25.2.2	<a href="#">Pattern match use case</a>	289
<b>25.3</b>	<b>Typedef Documentation</b>	<b>291</b>
25.3.1	<a href="#">pint_cb_t</a>	291
<b>25.4</b>	<b>Enumeration Type Documentation</b>	<b>292</b>
25.4.1	<a href="#">pint_pin_enable_t</a>	292
25.4.2	<a href="#">pint_pin_int_t</a>	292
25.4.3	<a href="#">pint_pmatch_input_src_t</a>	292
25.4.4	<a href="#">pint_pmatch_bslice_t</a>	292
25.4.5	<a href="#">pint_pmatch_bslice_cfg_t</a>	293
<b>25.5</b>	<b>Function Documentation</b>	<b>293</b>
25.5.1	<a href="#">PINT_Init</a>	293
25.5.2	<a href="#">PINT_PinInterruptConfig</a>	293
25.5.3	<a href="#">PINT_PinInterruptGetConfig</a>	294
25.5.4	<a href="#">PINT_PinInterruptClrStatus</a>	294
25.5.5	<a href="#">PINT_PinInterruptGetStatus</a>	294
25.5.6	<a href="#">PINT_PinInterruptClrStatusAll</a>	295
25.5.7	<a href="#">PINT_PinInterruptGetStatusAll</a>	295
25.5.8	<a href="#">PINT_PinInterruptClrFallFlag</a>	295
25.5.9	<a href="#">PINT_PinInterruptGetFallFlag</a>	296
25.5.10	<a href="#">PINT_PinInterruptClrFallFlagAll</a>	296
25.5.11	<a href="#">PINT_PinInterruptGetFallFlagAll</a>	296
25.5.12	<a href="#">PINT_PinInterruptClrRiseFlag</a>	297
25.5.13	<a href="#">PINT_PinInterruptGetRiseFlag</a>	297
25.5.14	<a href="#">PINT_PinInterruptClrRiseFlagAll</a>	298
25.5.15	<a href="#">PINT_PinInterruptGetRiseFlagAll</a>	298
25.5.16	<a href="#">PINT_PatternMatchConfig</a>	298

# Contents

Section Number	Title	Page Number
25.5.17	PINT_PatternMatchGetConfig . . . . .	299
25.5.18	PINT_PatternMatchGetStatus . . . . .	299
25.5.19	PINT_PatternMatchGetStatusAll . . . . .	300
25.5.20	PINT_PatternMatchResetDetectLogic . . . . .	300
25.5.21	PINT_PatternMatchEnable . . . . .	300
25.5.22	PINT_PatternMatchDisable . . . . .	301
25.5.23	PINT_PatternMatchEnableRXEV . . . . .	301
25.5.24	PINT_PatternMatchDisableRXEV . . . . .	301
25.5.25	PINT_EnableCallback . . . . .	302
25.5.26	PINT_DisableCallback . . . . .	302
25.5.27	PINT_Deinit . . . . .	302
<b>Chapter</b>	<b>CTIMER: Standard counter/timers</b>	
<b>26.1</b>	<b>Overview . . . . .</b>	<b>305</b>
<b>26.2</b>	<b>Function groups . . . . .</b>	<b>305</b>
26.2.1	Initialization and deinitialization . . . . .	305
26.2.2	PWM Operations . . . . .	305
26.2.3	Match Operation . . . . .	305
26.2.4	Input capture operations . . . . .	305
<b>26.3</b>	<b>Typical use case . . . . .</b>	<b>306</b>
26.3.1	Match example . . . . .	306
26.3.2	PWM output example . . . . .	306
<b>26.4</b>	<b>Data Structure Documentation . . . . .</b>	<b>308</b>
26.4.1	struct ctimer_match_config_t . . . . .	308
26.4.2	struct ctimer_config_t . . . . .	309
<b>26.5</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>309</b>
26.5.1	ctimer_capture_channel_t . . . . .	309
26.5.2	ctimer_capture_edge_t . . . . .	309
26.5.3	ctimer_match_t . . . . .	310
26.5.4	ctimer_match_output_control_t . . . . .	310
26.5.5	ctimer_interrupt_enable_t . . . . .	310
26.5.6	ctimer_status_flags_t . . . . .	310
26.5.7	ctimer_callback_type_t . . . . .	311
<b>26.6</b>	<b>Function Documentation . . . . .</b>	<b>311</b>
26.6.1	CTIMER_Init . . . . .	311
26.6.2	CTIMER_Deinit . . . . .	311
26.6.3	CTIMER_GetDefaultConfig . . . . .	311
26.6.4	CTIMER_SetupPwm . . . . .	312
26.6.5	CTIMER_UpdatePwmDutycycle . . . . .	312

# Contents

Section Number	Title	Page Number
26.6.6	CTIMER_SetupMatch . . . . .	313
26.6.7	CTIMER_SetupCapture . . . . .	313
26.6.8	CTIMER_RegisterCallBack . . . . .	313
26.6.9	CTIMER_EnableInterrupts . . . . .	314
26.6.10	CTIMER_DisableInterrupts . . . . .	314
26.6.11	CTIMER_GetEnabledInterrupts . . . . .	314
26.6.12	CTIMER_GetStatusFlags . . . . .	315
26.6.13	CTIMER_ClearStatusFlags . . . . .	316
26.6.14	CTIMER_StartTimer . . . . .	316
26.6.15	CTIMER_StopTimer . . . . .	316
26.6.16	CTIMER_Reset . . . . .	316
<b>Chapter</b>	<b>Common Driver</b>	
<b>27.1</b>	<b>Overview . . . . .</b>	<b>319</b>
<b>27.2</b>	<b>Macro Definition Documentation . . . . .</b>	<b>322</b>
27.2.1	MAKE_STATUS . . . . .	322
27.2.2	MAKE_VERSION . . . . .	322
27.2.3	FSL_COMMON_DRIVER_VERSION . . . . .	322
27.2.4	DEBUG_CONSOLE_DEVICE_TYPE_NONE . . . . .	322
27.2.5	DEBUG_CONSOLE_DEVICE_TYPE_UART . . . . .	322
27.2.6	DEBUG_CONSOLE_DEVICE_TYPE_LPUART . . . . .	322
27.2.7	DEBUG_CONSOLE_DEVICE_TYPE_LPSCI . . . . .	322
27.2.8	DEBUG_CONSOLE_DEVICE_TYPE_USBCDC . . . . .	322
27.2.9	DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM . . . . .	322
27.2.10	DEBUG_CONSOLE_DEVICE_TYPE_IUART . . . . .	322
27.2.11	DEBUG_CONSOLE_DEVICE_TYPE_VUSART . . . . .	322
27.2.12	DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART . . . . .	322
27.2.13	ARRAY_SIZE . . . . .	322
<b>27.3</b>	<b>Typedef Documentation . . . . .</b>	<b>323</b>
27.3.1	status_t . . . . .	323
<b>27.4</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>323</b>
27.4.1	_status_groups . . . . .	323
27.4.2	_generic_status . . . . .	324
<b>27.5</b>	<b>Function Documentation . . . . .</b>	<b>324</b>
27.5.1	EnableIRQ . . . . .	324
27.5.2	DisableIRQ . . . . .	326
27.5.3	DisableGlobalIRQ . . . . .	326
27.5.4	EnableGlobalIRQ . . . . .	326
27.5.5	EnableDeepSleepIRQ . . . . .	327
27.5.6	DisableDeepSleepIRQ . . . . .	327

# Contents

Section Number Chapter	Title	Page Number
	<b>Debug Console</b>	
<b>28.1</b>	<b>Overview</b>	<b>329</b>
<b>28.2</b>	<b>Function groups</b>	<b>329</b>
28.2.1	Initialization	329
28.2.2	Advanced Feature	330
<b>28.3</b>	<b>Typical use case</b>	<b>333</b>
<b>28.4</b>	<b>Data Structure Documentation</b>	<b>336</b>
28.4.1	struct io_state_t	336
<b>28.5</b>	<b>Macro Definition Documentation</b>	<b>336</b>
28.5.1	SDK_DEBUGCONSOLE	336
<b>28.6</b>	<b>Typedef Documentation</b>	<b>336</b>
28.6.1	notify	336
<b>28.7</b>	<b>Function Documentation</b>	<b>336</b>
28.7.1	DbgConsole_Init	336
28.7.2	DbgConsole_Deinit	337
28.7.3	DbgConsole_Printf	337
28.7.4	DbgConsole_Putchar	338
28.7.5	DbgConsole_Scanf	338
28.7.6	DbgConsole_Getchar	338
28.7.7	DbgConsole_Flush	339
28.7.8	IO_Init	339
28.7.9	IO_Deinit	339
28.7.10	IO_Transfer	339
28.7.11	IO_WaitIdle	340
28.7.12	LOG_Init	340
28.7.13	LOG_Deinit	341
28.7.14	LOG_Push	341
28.7.15	LOG_ReadLine	341
28.7.16	LOG_ReadCharacter	342
28.7.17	LOG_WaitIdle	342
28.7.18	LOG_Pop	342
28.7.19	StrFormatPrintf	342
28.7.20	StrFormatScanf	343
<b>28.8</b>	<b>Semihosting</b>	<b>344</b>
28.8.1	Guide Semihosting for IAR	344
28.8.2	Guide Semihosting for Keil $\mu$ Vision	344
28.8.3	Guide Semihosting for KDS	346
28.8.4	Guide Semihosting for MCUX	346

# Contents

Section Number	Title	Page Number
28.8.5	Guide Semihosting for ARMGCC . . . . .	347



# Chapter 1

## Introduction

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

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

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

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

- IAR Embedded Workbench
- Keil MDK
- MCUXpresso IDE

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

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

<b>Deliverable</b>	<b>Location</b>
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_SPI_Busy` = 1400
- `kStatus_SPI_Idle` = 1401
- `kStatus_SPI_Error` = 1402
- `kStatus_DMA_Busy` = 5000



## Chapter 3

### 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://nxp.com)

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

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

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

© 2017 NXP B.V.



## Chapter 4

### Architectural Overview

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

#### Overview

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

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

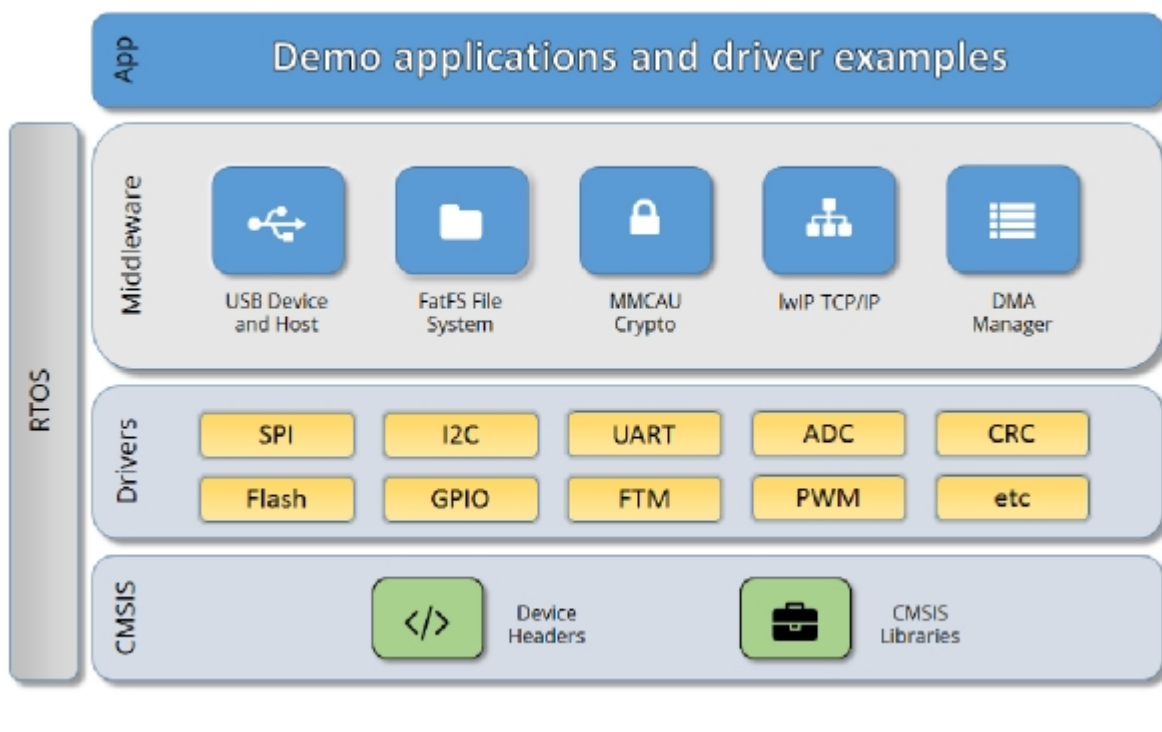


Figure 1: MCUXpresso SDK Block Diagram

#### MCU header files

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

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

## CMSIS Support

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

## MCUXpresso SDK Peripheral Drivers

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

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

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

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

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

## Interrupt handling for transactional APIs

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

```
PUBWEAK SPI0_IRQHandler
PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
```

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

The first level of the weak implementation are the functions defined in the vector table. In the devices/(<DEVICE\_NAME>/(<TOOLCHAIN>/startup\_<DEVICE\_NAME>.s/.S file, the implementation of the first layer weak function calls the second layer of weak function. The implementation of the second layer weak function (ex. SPI0\_DriverIRQHandler) jumps to itself (B .). The MCUXpresso SDK drivers with transactional APIs provide the reimplement of the second layer function inside of the peripheral driver. If the MCUXpresso SDK drivers with transactional APIs are linked into the image, the SPI0\_DriverIRQHandler is replaced with the function implemented in the MCUXpresso SDK SPI driver.

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

The limitation of the double weak mechanism is that it cannot be used for peripherals that share the same vector entry. For this use case, redefine the first layer weak function to enable the desired peripheral interrupt functionality. For example, if the MCU's UART0 and UART1 share the same vector entry, redefine the UART0\_UART1\_IRQHandler according to the use case requirements.

### Feature Header Files

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

### Application

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







## Chapter 5

### FLEXCOMM: FLEXCOMM Driver

#### 5.1 Overview

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

#### Modules

- [FLEXCOMM Driver](#)

## **5.2 FLEXCOMM Driver**

## Chapter 6

# I2C: Inter-Integrated Circuit Driver

### 6.1 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 are feature/property target 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 the knowledge of 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 are transaction target high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code 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.

### 6.2 Typical use case

#### 6.2.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* Get default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Init I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

/* Send start and slave address. */
I2C_MasterStart(EXAMPLE_I2C_MASTER_BASEADDR, 7-bit slave address,
    kI2C_Write/kI2C_Read);

/* Wait address sent out. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR)) & kI2C_IntPendingFlag))
{
}

if(status & kI2C_ReceiveNakFlag)
{
    return kStatus_I2C_Nak;
}
```

## Typical use case

```
result = I2C_MasterWriteBlocking(EXAMPLE_I2C_MASTER_BASEADDR, txBuff, BUFFER_SIZE);

if(result)
{
    /* If error occurs, send STOP. */
    I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
    return result;
}

while(!(I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR) & kI2C_IntPendingFlag))
{

}

/* Wait all data sent out, send STOP. */
I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
```

### 6.2.2 Master Operation in interrupt transactional method

```
i2c_master_handle_t g_m_handle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_handle_t *handle,
    status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Get default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Init I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

I2C_MasterTransferCreateHandle(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle,
    i2c_master_callback, NULL);
I2C_MasterTransferNonBlocking(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle, &
    masterXfer);

/* Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

### 6.2.3 Master Operation in DMA transactional method

```

i2c_master_dma_handle_t g_m_dma_handle;
dma_handle_t dmaHandle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_dma_handle_t *handle,
    status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Get default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Init I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

DMA_EnableChannel(EXAMPLE_DMA, EXAMPLE_I2C_MASTER_CHANNEL);
DMA_CreateHandle(&dmaHandle, EXAMPLE_DMA, EXAMPLE_I2C_MASTER_CHANNEL);

I2C_MasterTransferCreateHandleDMA(EXAMPLE_I2C_MASTER_BASEADDR, &
    g_m_dma_handle, i2c_master_callback, NULL, &dmaHandle);
I2C_MasterTransferDMA(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_dma_handle, &masterXfer);

/* Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;

```

### 6.2.4 Slave Operation in functional method

```

i2c_slave_config_t slaveConfig;
uint8_t status;
status_t result = kStatus_Success;

I2C_SlaveGetDefaultConfig(&slaveConfig); /*default configuration 7-bit addressing
    mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/kI2C_RangeMatch;
I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

/* Wait address match. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_SLAVE_BASEADDR)) & kI2C_AddressMatchFlag))
{
}

```

## Typical use case

```
/* Slave transmit, master reading from slave. */
if (status & kI2C_TransferDirectionFlag)
{
    result = I2C_SlaveWriteBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}
else
{
    I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}

return result;
```

### 6.2.5 Slave Operation in interrupt transactional method

```
i2c_slave_config_t slaveConfig;
i2c_slave_handle_t g_s_handle;
volatile bool g_SlaveCompletionFlag = false;

static void i2c_slave_callback(I2C_Type *base, i2c_slave_transfer_t *xfer, void *
    userData)
{
    switch (xfer->event)
    {
        /* Transmit request */
        case kI2C_SlaveTransmitEvent:
            /* Update information for transmit process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Receive request */
        case kI2C_SlaveReceiveEvent:
            /* Update information for received process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Transfer done */
        case kI2C_SlaveCompletionEvent:
            g_SlaveCompletionFlag = true;
            break;

        default:
            g_SlaveCompletionFlag = true;
            break;
    }
}

I2C_SlaveGetDefaultConfig(&slaveConfig); /*default configuration 7-bit addressing
    mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/kI2C_RangeMatch;

I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig);

I2C_SlaveTransferCreateHandle(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
    i2c_slave_callback, NULL);

I2C_SlaveTransferNonBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
    kI2C_SlaveCompletionEvent);

/* Wait for transfer completed. */
while (!g_SlaveCompletionFlag)
{
}
```

```
g_SlaveCompletionFlag = false;
```

## Modules

- [I2C DMA Driver](#)
- [I2C Driver](#)
- [I2C FreeRTOS Driver](#)
- [I2C Master Driver](#)
- [I2C Slave Driver](#)

## I2C Driver

### 6.3 I2C Driver

#### 6.3.1 Overview

#### Files

- file [fsl\\_i2c.h](#)

#### Macros

- `#define I2C_WAIT_TIMEOUT 0U` /\* Define to zero means keep waiting until the flag is asserted/deassert. \*/  
*Timeout times for waiting flag.*
- `#define I2C_STAT_MSTCODE_IDLE (0)`  
*Master Idle State Code.*
- `#define I2C_STAT_MSTCODE_RXREADY (1)`  
*Master Receive Ready State Code.*
- `#define I2C_STAT_MSTCODE_TXREADY (2)`  
*Master Transmit Ready State Code.*
- `#define I2C_STAT_MSTCODE_NACKADR (3)`  
*Master NACK by slave on address State Code.*
- `#define I2C_STAT_MSTCODE_NACKDAT (4)`  
*Master NACK by slave on data State Code.*

#### Enumerations

- `enum _i2c_status {`  
`kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 0),`  
`kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 1),`  
`kStatus_I2C_Nak,`  
`kStatus_I2C_InvalidParameter,`  
`kStatus_I2C_BitError = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 4),`  
`kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 5),`  
`kStatus_I2C_NoTransferInProgress,`  
`kStatus_I2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 7) ,`  
`kStatus_I2C_Timeout = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 10) }`  
*I2C status return codes.*

#### Driver version

- `#define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))`  
*I2C driver version 2.0.2.*



## 6.3.2 Macro Definition Documentation

6.3.2.1 **#define FSL\_I2C\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 2))**

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

## 6.3.3 Enumeration Type Documentation

### 6.3.3.1 enum \_i2c\_status

Enumerator

***kStatus\_I2C\_Busy*** The master is already performing a transfer.

***kStatus\_I2C\_Idle*** The slave driver is idle.

***kStatus\_I2C\_Nak*** The slave device sent a NAK in response to a byte.

***kStatus\_I2C\_InvalidParameter*** Unable to proceed due to invalid parameter.

***kStatus\_I2C\_BitError*** Transferred bit was not seen on the bus.

***kStatus\_I2C\_ArbitrationLost*** Arbitration lost error.

***kStatus\_I2C\_NoTransferInProgress*** Attempt to abort a transfer when one is not in progress.

***kStatus\_I2C\_DmaRequestFail*** DMA request failed.

***kStatus\_I2C\_Timeout*** Timeout polling status flags.

## I2C Master Driver

### 6.4 I2C Master Driver

#### 6.4.1 Overview

#### Data Structures

- struct [i2c\\_master\\_config\\_t](#)  
*Structure with settings to initialize the I2C master module. [More...](#)*
- struct [i2c\\_master\\_transfer\\_t](#)  
*Non-blocking transfer descriptor structure. [More...](#)*
- struct [i2c\\_master\\_handle\\_t](#)  
*Driver handle for master non-blocking APIs. [More...](#)*

#### Typedefs

- typedef void(\* [i2c\\_master\\_transfer\\_callback\\_t](#) )(I2C\_Type \*base, [i2c\\_master\\_handle\\_t](#) \*handle, [status\\_t](#) completionStatus, void \*userData)  
*Master completion callback function pointer type.*

#### Enumerations

- enum [\\_i2c\\_master\\_flags](#) {  
    [kI2C\\_MasterPendingFlag](#) = I2C\_STAT\_MSTPENDING\_MASK,  
    [kI2C\\_MasterArbitrationLostFlag](#),  
    [kI2C\\_MasterStartStopErrorFlag](#) }  
*I2C master peripheral flags.*
- enum [i2c\\_direction\\_t](#) {  
    [kI2C\\_Write](#) = 0U,  
    [kI2C\\_Read](#) = 1U }  
*Direction of master and slave transfers.*
- enum [\\_i2c\\_master\\_transfer\\_flags](#) {  
    [kI2C\\_TransferDefaultFlag](#) = 0x00U,  
    [kI2C\\_TransferNoStartFlag](#) = 0x01U,  
    [kI2C\\_TransferRepeatedStartFlag](#) = 0x02U,  
    [kI2C\\_TransferNoStopFlag](#) = 0x04U }  
*Transfer option flags.*
- enum [\\_i2c\\_transfer\\_states](#)  
*States for the state machine used by transactional APIs.*

#### Initialization and deinitialization

- void [I2C\\_MasterGetDefaultConfig](#) ([i2c\\_master\\_config\\_t](#) \*masterConfig)  
*Provides a default configuration for the I2C master peripheral.*
- void [I2C\\_MasterInit](#) (I2C\_Type \*base, const [i2c\\_master\\_config\\_t](#) \*masterConfig, uint32\_t src-Clock\_Hz)

- *Initializes the I2C master peripheral.*
- void [I2C\\_MasterDeinit](#) (I2C\_Type \*base)
- *Deinitializes the I2C master peripheral.*
- static void [I2C\\_MasterReset](#) (I2C\_Type \*base)
- *Performs a software reset.*
- static void [I2C\\_MasterEnable](#) (I2C\_Type \*base, bool enable)
- *Enables or disables the I2C module as master.*

## Status

- static uint32\_t [I2C\\_GetStatusFlags](#) (I2C\_Type \*base)
- *Gets the I2C status flags.*
- static void [I2C\\_MasterClearStatusFlags](#) (I2C\_Type \*base, uint32\_t statusMask)
- *Clears the I2C master status flag state.*

## Interrupts

- static void [I2C\\_EnableInterrupts](#) (I2C\_Type \*base, uint32\_t interruptMask)
- *Enables the I2C master interrupt requests.*
- static void [I2C\\_DisableInterrupts](#) (I2C\_Type \*base, uint32\_t interruptMask)
- *Disables the I2C master interrupt requests.*
- static uint32\_t [I2C\\_GetEnabledInterrupts](#) (I2C\_Type \*base)
- *Returns the set of currently enabled I2C master interrupt requests.*

## Bus operations

- void [I2C\\_MasterSetBaudRate](#) (I2C\_Type \*base, uint32\_t baudRate\_Bps, uint32\_t srcClock\_Hz)
- *Sets the I2C bus frequency for master transactions.*
- static bool [I2C\\_MasterGetBusIdleState](#) (I2C\_Type \*base)
- *Returns whether the bus is idle.*
- [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.*
- static [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 void \*txBuff, size\_t txSize, uint32\_t flags)
- *Performs a polling send transfer on the I2C bus.*
- [status\\_t](#) [I2C\\_MasterReadBlocking](#) (I2C\_Type \*base, void \*rxBuff, size\_t rxSize, uint32\_t flags)
- *Performs a polling receive transfer 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.*

## I2C Master Driver

### Non-blocking

- void [I2C\\_MasterTransferCreateHandle](#) (I2C\_Type \*base, i2c\_master\_handle\_t \*handle, [i2c\\_master\\_transfer\\_callback\\_t](#) callback, void \*userData)  
*Creates a new handle for the I2C master non-blocking APIs.*
- [status\\_t I2C\\_MasterTransferNonBlocking](#) (I2C\_Type \*base, i2c\_master\_handle\_t \*handle, [i2c\\_master\\_transfer\\_t](#) \*xfer)  
*Performs a non-blocking transaction on the I2C bus.*
- [status\\_t I2C\\_MasterTransferGetCount](#) (I2C\_Type \*base, i2c\_master\_handle\_t \*handle, size\_t \*count)  
*Returns number of bytes transferred so far.*
- [status\\_t I2C\\_MasterTransferAbort](#) (I2C\_Type \*base, i2c\_master\_handle\_t \*handle)  
*Terminates a non-blocking I2C master transmission early.*

### IRQ handler

- void [I2C\\_MasterTransferHandleIRQ](#) (I2C\_Type \*base, i2c\_master\_handle\_t \*handle)  
*Reusable routine to handle master interrupts.*

## 6.4.2 Data Structure Documentation

### 6.4.2.1 struct i2c\_master\_config\_t

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the [I2C\\_MasterGetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

### Data Fields

- bool [enableMaster](#)  
*Whether to enable master mode.*
- uint32\_t [baudRate\\_Bps](#)  
*Desired baud rate in bits per second.*
- bool [enableTimeout](#)  
*Enable internal timeout function.*

#### 6.4.2.1.0.1 Field Documentation

6.4.2.1.0.1.1 `bool i2c_master_config_t::enableMaster`

6.4.2.1.0.1.2 `uint32_t i2c_master_config_t::baudRate_Bps`

6.4.2.1.0.1.3 `bool i2c_master_config_t::enableTimeout`

#### 6.4.2.2 `struct _i2c_master_transfer`

I2C master transfer typedef.

This structure is used to pass transaction parameters to the [I2C\\_MasterTransferNonBlocking\(\)](#) API.

#### Data Fields

- `uint32_t flags`  
*Bit mask of options for the transfer.*
- `uint16_t slaveAddress`  
*The 7-bit slave address.*
- `i2c_direction_t direction`  
*Either [kI2C\\_Read](#) or [kI2C\\_Write](#).*
- `uint32_t subaddress`  
*Sub address.*
- `size_t subaddressSize`  
*Length of sub address to send in bytes.*
- `void * data`  
*Pointer to data to transfer.*
- `size_t dataSize`  
*Number of bytes to transfer.*

#### 6.4.2.2.0.2 Field Documentation

6.4.2.2.0.2.1 `uint32_t i2c_master_transfer_t::flags`

See enumeration [\\_i2c\\_master\\_transfer\\_flags](#) for available options. Set to 0 or [kI2C\\_TransferDefaultFlag](#) for normal transfers.

6.4.2.2.0.2.2 `uint16_t i2c_master_transfer_t::slaveAddress`

6.4.2.2.0.2.3 `i2c_direction_t i2c_master_transfer_t::direction`

6.4.2.2.0.2.4 `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

6.4.2.2.0.2.5 `size_t i2c_master_transfer_t::subaddressSize`

Maximum size is 4 bytes.

## I2C Master Driver

6.4.2.2.0.2.6 void\* i2c\_master\_transfer\_t::data

6.4.2.2.0.2.7 size\_t i2c\_master\_transfer\_t::dataSize

### 6.4.2.3 struct\_i2c\_master\_handle

I2C master handle typedef.

Note

The contents of this structure are private and subject to change.

#### Data Fields

- uint8\_t [state](#)  
*Transfer state machine current state.*
- uint32\_t [transferCount](#)  
*Indicates progress of the transfer.*
- uint32\_t [remainingBytes](#)  
*Remaining byte count in current state.*
- uint8\_t \* [buf](#)  
*Buffer pointer for current state.*
- i2c\_master\_transfer\_t [transfer](#)  
*Copy of the current transfer info.*
- [i2c\\_master\\_transfer\\_callback\\_t](#) [completionCallback](#)  
*Callback function pointer.*
- void \* [userData](#)  
*Application data passed to callback.*

#### 6.4.2.3.0.3 Field Documentation

6.4.2.3.0.3.1 `uint8_t i2c_master_handle_t::state`

6.4.2.3.0.3.2 `uint32_t i2c_master_handle_t::remainingBytes`

6.4.2.3.0.3.3 `uint8_t* i2c_master_handle_t::buf`

6.4.2.3.0.3.4 `i2c_master_transfer_t i2c_master_handle_t::transfer`

6.4.2.3.0.3.5 `i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback`

6.4.2.3.0.3.6 `void* i2c_master_handle_t::userData`

#### 6.4.3 Typedef Documentation

6.4.3.1 `typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base,  
i2c_master_handle_t *handle, status_t completionStatus, void *userData)`

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to [I2C\\_MasterTransferCreateHandle\(\)](#).

## I2C Master Driver

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>completion-Status</i>	Either kStatus_Success or an error code describing how the transfer completed.
<i>userData</i>	Arbitrary pointer-sized value passed from the application.

## 6.4.4 Enumeration Type Documentation

### 6.4.4.1 enum \_i2c\_master\_flags

#### Note

These enums are meant to be OR'd together to form a bit mask.

#### Enumerator

***kI2C\_MasterPendingFlag*** The I2C module is waiting for software interaction.

***kI2C\_MasterArbitrationLostFlag*** The arbitration of the bus was lost. There was collision on the bus

***kI2C\_MasterStartStopErrorFlag*** There was an error during start or stop phase of the transaction.

### 6.4.4.2 enum i2c\_direction\_t

#### Enumerator

***kI2C\_Write*** Master transmit.

***kI2C\_Read*** Master receive.

### 6.4.4.3 enum \_i2c\_master\_transfer\_flags

#### Note

These enumerations are intended to be OR'd together to form a bit mask of options for the [\\_i2c\\_master\\_transfer::flags](#) field.

#### Enumerator

***kI2C\_TransferDefaultFlag*** Transfer starts with a start signal, stops with a stop signal.

***kI2C\_TransferNoStartFlag*** Don't send a start condition, address, and sub address.

***kI2C\_TransferRepeatedStartFlag*** Send a repeated start condition.

***kI2C\_TransferNoStopFlag*** Don't send a stop condition.



#### 6.4.4.4 enum `_i2c_transfer_states`

### 6.4.5 Function Documentation

#### 6.4.5.1 void `I2C_MasterGetDefaultConfig ( i2c_master_config_t * masterConfig )`

This function provides the following default configuration for the I2C master peripheral:

```
* masterConfig->enableMaster      = true;
* masterConfig->baudRate_Bps      = 100000U;
* masterConfig->enableTimeout     = false;
*
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I2C_MasterInit()`.

Parameters

out	<i>masterConfig</i>	User provided configuration structure for default values. Refer to <a href="#">i2c_master_config_t</a> .
-----	---------------------	--

#### 6.4.5.2 void `I2C_MasterInit ( I2C_Type * base, const i2c_master_config_t * masterConfig, uint32_t srcClock_Hz )`

This function enables the peripheral clock and initializes the I2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>masterConfig</i>	User provided peripheral configuration. Use <a href="#">I2C_MasterGetDefaultConfig()</a> to get a set of defaults that you can override.
<i>srcClock_Hz</i>	Frequency in Hertz of the I2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

#### 6.4.5.3 void `I2C_MasterDeinit ( I2C_Type * base )`

This function disables the I2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

## I2C Master Driver

### Parameters

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

#### 6.4.5.4 static void I2C\_MasterReset ( I2C\_Type \* *base* ) [inline], [static]

Restores the I2C master peripheral to reset conditions.

### Parameters

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

#### 6.4.5.5 static void I2C\_MasterEnable ( I2C\_Type \* *base*, bool *enable* ) [inline], [static]

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>enable</i>	Pass true to enable or false to disable the specified I2C as master.

#### 6.4.5.6 static uint32\_t I2C\_GetStatusFlags ( I2C\_Type \* *base* ) [inline], [static]

A bit mask with the state of all I2C status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

### Parameters

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

### Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

### See Also

[\\_i2c\\_master\\_flags](#)

#### 6.4.5.7 static void I2C\_MasterClearStatusFlags ( I2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared:

- [kI2C\\_MasterArbitrationLostFlag](#)
- [kI2C\\_MasterStartStopErrorFlag](#)

Attempts to clear other flags has no effect.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>statusMask</i>	A bitmask of status flags that are to be cleared. The mask is composed of <a href="#">_i2c_master_flags</a> enumerators OR'd together. You may pass the result of a previous call to <a href="#">I2C_GetStatusFlags()</a> .

See Also

[\\_i2c\\_master\\_flags](#).

#### 6.4.5.8 static void I2C\_EnableInterrupts ( I2C\_Type \* *base*, uint32\_t *interruptMask* ) [inline], [static]

Parameters

<i>base</i>	The I2C peripheral base address.
<i>interruptMask</i>	Bit mask of interrupts to enable. See <a href="#">_i2c_master_flags</a> for the set of constants that should be OR'd together to form the bit mask.

#### 6.4.5.9 static void I2C\_DisableInterrupts ( I2C\_Type \* *base*, uint32\_t *interruptMask* ) [inline], [static]

Parameters

<i>base</i>	The I2C peripheral base address.
<i>interruptMask</i>	Bit mask of interrupts to disable. See <a href="#">_i2c_master_flags</a> for the set of constants that should be OR'd together to form the bit mask.

#### 6.4.5.10 static uint32\_t I2C\_GetEnabledInterrupts ( I2C\_Type \* *base* ) [inline], [static]

## I2C Master Driver

### Parameters

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

### Returns

A bitmask composed of [\\_i2c\\_master\\_flags](#) enumerators OR'd together to indicate the set of enabled interrupts.

#### 6.4.5.11 void I2C\_MasterSetBaudRate ( I2C\_Type \* *base*, uint32\_t *baudRate\_Bps*, uint32\_t *srcClock\_Hz* )

The I2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>srcClock_Hz</i>	I2C functional clock frequency in Hertz.
<i>baudRate_Bps</i>	Requested bus frequency in bits per second.

#### 6.4.5.12 static bool I2C\_MasterGetBusIdleState ( I2C\_Type \* *base* ) [inline], [static]

Requires the master mode to be enabled.

### Parameters

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

### Return values

<i>true</i>	Bus is busy.
<i>false</i>	Bus is idle.

#### 6.4.5.13 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.

**6.4.5.14 status\_t I2C\_MasterStop ( I2C\_Type \* *base* )**

## Return values

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

**6.4.5.15 static status\_t I2C\_MasterRepeatedStart ( I2C\_Type \* *base*, uint8\_t *address*, i2c\_direction\_t *direction* ) [inline], [static]**

## 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.

**6.4.5.16 status\_t I2C\_MasterWriteBlocking ( I2C\_Type \* *base*, const void \* *txBuff*, size\_t *txSize*, uint32\_t *flags* )**

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns [kStatus\\_I2C\\_Nak](#).

## I2C Master Driver

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>txBuff</i>	The pointer to the data to be transferred.
<i>txSize</i>	The length in bytes of the data to be transferred.
<i>flags</i>	Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

### Return values

<i>kStatus_Success</i>	Data was sent successfully.
<i>kStatus_I2C_Busy</i>	Another master is currently utilizing the bus.
<i>kStatus_I2C_Nak</i>	The slave device sent a NAK in response to a byte.
<i>kStatus_I2C_Arbitration-Lost</i>	Arbitration lost error.

#### 6.4.5.17 status\_t I2C\_MasterReadBlocking ( I2C\_Type \* *base*, void \* *rxBuff*, size\_t *rxSize*, uint32\_t *flags* )

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>rxBuff</i>	The pointer to the data to be transferred.
<i>rxSize</i>	The length in bytes of the data to be transferred.
<i>flags</i>	Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

### Return values

<i>kStatus_Success</i>	Data was received successfully.
<i>kStatus_I2C_Busy</i>	Another master is currently utilizing the bus.
<i>kStatus_I2C_Nak</i>	The slave device sent a NAK in response to a byte.
<i>kStatus_I2C_Arbitration-Lost</i>	Arbitration lost error.

#### 6.4.5.18 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.

#### 6.4.5.19 void I2C\_MasterTransferCreateHandle ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle*, i2c\_master\_transfer\_callback\_t *callback*, void \* *userData* )

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the [I2C\\_MasterTransferAbort\(\)](#) API shall be called.

## Parameters

	<i>base</i>	The I2C peripheral base address.
out	<i>handle</i>	Pointer to the I2C master driver handle.
	<i>callback</i>	User provided pointer to the asynchronous callback function.
	<i>userData</i>	User provided pointer to the application callback data.

#### 6.4.5.20 status\_t I2C\_MasterTransferNonBlocking ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle*, i2c\_master\_transfer\_t \* *xfer* )

## I2C Master Driver

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to the I2C master driver handle.
<i>xfer</i>	The pointer to the transfer descriptor.

### Return values

<i>kStatus_Success</i>	The transaction was started successfully.
<i>kStatus_I2C_Busy</i>	Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

#### 6.4.5.21 **status\_t I2C\_MasterTransferGetCount ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle*, size\_t \* *count* )**

### Parameters

	<i>base</i>	The I2C peripheral base address.
	<i>handle</i>	Pointer to the I2C master driver handle.
out	<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

### Return values

<i>kStatus_Success</i>	
<i>kStatus_I2C_Busy</i>	

#### 6.4.5.22 **status\_t I2C\_MasterTransferAbort ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle* )**

### Note

It is not safe to call this function from an IRQ handler that has a higher priority than the I2C peripheral's IRQ priority.

### Parameters



<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to the I2C master driver handle.

Return values

<i>kStatus_Success</i>	A transaction was successfully aborted.
<i>kStatus_I2C_Timeout</i>	Timeout during polling for flags.

#### 6.4.5.23 void I2C\_MasterTransferHandleIRQ ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle* )

Note

This function does not need to be called unless you are reimplementing the nonblocking API's interrupt handler routines to add special functionality.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to the I2C master driver handle.

### 6.5 I2C Slave Driver

#### 6.5.1 Overview

#### Data Structures

- struct [i2c\\_slave\\_address\\_t](#)  
*Data structure with 7-bit Slave address and Slave address disable. [More...](#)*
- struct [i2c\\_slave\\_config\\_t](#)  
*Structure with settings to initialize the I2C slave module. [More...](#)*
- struct [i2c\\_slave\\_transfer\\_t](#)  
*I2C slave transfer structure. [More...](#)*
- struct [i2c\\_slave\\_handle\\_t](#)  
*I2C slave handle structure. [More...](#)*

#### Typedefs

- typedef void(\* [i2c\\_slave\\_transfer\\_callback\\_t](#) )(I2C\_Type \*base, volatile [i2c\\_slave\\_transfer\\_t](#) \*transfer, void \*userData)  
*Slave event callback function pointer type.*

#### Enumerations

- enum [\\_i2c\\_slave\\_flags](#) {  
    [kI2C\\_SlavePendingFlag](#) = I2C\_STAT\_SLVPENDING\_MASK,  
    [kI2C\\_SlaveNotStretching](#),  
    [kI2C\\_SlaveSelected](#) = I2C\_STAT\_SLVSEL\_MASK,  
    [kI2C\\_SaveDeselected](#) }  
*I2C slave peripheral flags.*
- enum [i2c\\_slave\\_address\\_register\\_t](#) {  
    [kI2C\\_SlaveAddressRegister0](#) = 0U,  
    [kI2C\\_SlaveAddressRegister1](#) = 1U,  
    [kI2C\\_SlaveAddressRegister2](#) = 2U,  
    [kI2C\\_SlaveAddressRegister3](#) = 3U }  
*I2C slave address register.*
- enum [i2c\\_slave\\_address\\_qual\\_mode\\_t](#) {  
    [kI2C\\_QualModeMask](#) = 0U,  
    [kI2C\\_QualModeExtend](#) }  
*I2C slave address match options.*
- enum [i2c\\_slave\\_bus\\_speed\\_t](#)  
*I2C slave bus speed options.*
- enum [i2c\\_slave\\_transfer\\_event\\_t](#) {

```

kI2C_SlaveAddressMatchEvent = 0x01U,
kI2C_SlaveTransmitEvent = 0x02U,
kI2C_SlaveReceiveEvent = 0x04U,
kI2C_SlaveCompletionEvent = 0x20U,
kI2C_SlaveDeselectedEvent,
kI2C_SlaveAllEvents }

```

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

- enum `i2c_slave_fsm_t`

*I2C slave software finite state machine states.*

## Slave initialization and deinitialization

- void `I2C_SlaveGetDefaultConfig` (`i2c_slave_config_t` \*slaveConfig)  
*Provides a default configuration for the I2C slave peripheral.*
- `status_t I2C_SlaveInit` (`I2C_Type` \*base, const `i2c_slave_config_t` \*slaveConfig, `uint32_t` srcClock-  
\_Hz)  
*Initializes the I2C slave peripheral.*
- void `I2C_SlaveSetAddress` (`I2C_Type` \*base, `i2c_slave_address_register_t` addressRegister, `uint8_t`  
address, bool addressDisable)  
*Configures Slave Address n register.*
- void `I2C_SlaveDeinit` (`I2C_Type` \*base)  
*Deinitializes the I2C slave peripheral.*
- static void `I2C_SlaveEnable` (`I2C_Type` \*base, bool enable)  
*Enables or disables the I2C module as slave.*

## Slave status

- static void `I2C_SlaveClearStatusFlags` (`I2C_Type` \*base, `uint32_t` statusMask)  
*Clears the I2C status flag state.*

## Slave bus operations

- `status_t I2C_SlaveWriteBlocking` (`I2C_Type` \*base, const `uint8_t` \*txBuff, `size_t` txSize)  
*Performs a polling send transfer on the I2C bus.*
- `status_t I2C_SlaveReadBlocking` (`I2C_Type` \*base, `uint8_t` \*rxBuff, `size_t` rxSize)  
*Performs a polling receive transfer on the I2C bus.*

## Slave non-blocking

- void `I2C_SlaveTransferCreateHandle` (`I2C_Type` \*base, `i2c_slave_handle_t` \*handle, `i2c_slave_-`  
`transfer_callback_t` callback, void \*userData)  
*Creates a new handle for the I2C slave non-blocking APIs.*
- `status_t I2C_SlaveTransferNonBlocking` (`I2C_Type` \*base, `i2c_slave_handle_t` \*handle, `uint32_t`  
eventMask)

## I2C Slave Driver

- Starts accepting slave transfers.*
- [status\\_t I2C\\_SlaveSetSendBuffer](#) (I2C\_Type \*base, volatile [i2c\\_slave\\_transfer\\_t](#) \*transfer, const void \*txData, size\_t txSize, uint32\_t eventMask)  
*Starts accepting master read from slave requests.*
- [status\\_t I2C\\_SlaveSetReceiveBuffer](#) (I2C\_Type \*base, volatile [i2c\\_slave\\_transfer\\_t](#) \*transfer, void \*rxData, size\_t rxSize, uint32\_t eventMask)  
*Starts accepting master write to slave requests.*
- static [uint32\\_t I2C\\_SlaveGetReceivedAddress](#) (I2C\_Type \*base, volatile [i2c\\_slave\\_transfer\\_t](#) \*transfer)  
*Returns the slave address sent by the I2C master.*
- void [I2C\\_SlaveTransferAbort](#) (I2C\_Type \*base, [i2c\\_slave\\_handle\\_t](#) \*handle)  
*Aborts the slave non-blocking transfers.*
- [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.*

## Slave IRQ handler

- void [I2C\\_SlaveTransferHandleIRQ](#) (I2C\_Type \*base, [i2c\\_slave\\_handle\\_t](#) \*handle)  
*Reusable routine to handle slave interrupts.*

## 6.5.2 Data Structure Documentation

### 6.5.2.1 struct i2c\_slave\_address\_t

#### Data Fields

- [uint8\\_t address](#)  
*7-bit Slave address SLVADR.*
- [bool addressDisable](#)  
*Slave address disable SADISABLE.*

#### 6.5.2.1.0.4 Field Documentation

##### 6.5.2.1.0.4.1 [uint8\\_t i2c\\_slave\\_address\\_t::address](#)

##### 6.5.2.1.0.4.2 [bool i2c\\_slave\\_address\\_t::addressDisable](#)

### 6.5.2.2 struct i2c\_slave\_config\_t

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the [I2C\\_SlaveGetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

## Data Fields

- [i2c\\_slave\\_address\\_t address0](#)  
*Slave's 7-bit address and disable.*
- [i2c\\_slave\\_address\\_t address1](#)  
*Alternate slave 7-bit address and disable.*
- [i2c\\_slave\\_address\\_t address2](#)  
*Alternate slave 7-bit address and disable.*
- [i2c\\_slave\\_address\\_t address3](#)  
*Alternate slave 7-bit address and disable.*
- [i2c\\_slave\\_address\\_qual\\_mode\\_t qualMode](#)  
*Qualify mode for slave address 0.*
- [uint8\\_t qualAddress](#)  
*Slave address qualifier for address 0.*
- [i2c\\_slave\\_bus\\_speed\\_t busSpeed](#)  
*Slave bus speed mode.*
- [bool enableSlave](#)  
*Enable slave mode.*

### 6.5.2.2.0.5 Field Documentation

**6.5.2.2.0.5.1 [i2c\\_slave\\_address\\_t i2c\\_slave\\_config\\_t::address0](#)**

**6.5.2.2.0.5.2 [i2c\\_slave\\_address\\_t i2c\\_slave\\_config\\_t::address1](#)**

**6.5.2.2.0.5.3 [i2c\\_slave\\_address\\_t i2c\\_slave\\_config\\_t::address2](#)**

**6.5.2.2.0.5.4 [i2c\\_slave\\_address\\_t i2c\\_slave\\_config\\_t::address3](#)**

**6.5.2.2.0.5.5 [i2c\\_slave\\_address\\_qual\\_mode\\_t i2c\\_slave\\_config\\_t::qualMode](#)**

**6.5.2.2.0.5.6 [uint8\\_t i2c\\_slave\\_config\\_t::qualAddress](#)**

**6.5.2.2.0.5.7 [i2c\\_slave\\_bus\\_speed\\_t i2c\\_slave\\_config\\_t::busSpeed](#)**

If the slave function stretches SCL to allow for software response, it must provide sufficient data setup time to the master before releasing the stretched clock. This is accomplished by inserting one clock time of CLKDIV at that point. The [busSpeed](#) value is used to configure CLKDIV such that one clock time is greater than the tSU;DAT value noted in the I2C bus specification for the I2C mode that is being used. If the [busSpeed](#) mode is unknown at compile time, use the longest data setup time [kI2C\\_SlaveStandardMode](#) (250 ns)

**6.5.2.2.0.5.8 [bool i2c\\_slave\\_config\\_t::enableSlave](#)**

### 6.5.2.3 [struct i2c\\_slave\\_transfer\\_t](#)

## Data Fields

- [i2c\\_slave\\_handle\\_t \\* handle](#)  
*Pointer to handle that contains this transfer.*
- [i2c\\_slave\\_transfer\\_event\\_t event](#)

## I2C Slave Driver

- *Reason the callback is being invoked.*  
uint8\_t [receivedAddress](#)
- *Matching address send by master.*  
uint32\_t [eventMask](#)
- *Mask of enabled events.*  
uint8\_t \* [rxData](#)
- *Transfer buffer for receive data.*  
const uint8\_t \* [txData](#)
- *Transfer buffer for transmit data.*  
size\_t [txSize](#)
- *Transfer size.*  
size\_t [rxSize](#)
- *Transfer size.*  
size\_t [transferredCount](#)
- *Number of bytes transferred during this transfer.*  
[status\\_t](#) [completionStatus](#)
- *Success or error code describing how the transfer completed.*

### 6.5.2.3.0.6 Field Documentation

**6.5.2.3.0.6.1 i2c\_slave\_handle\_t\* i2c\_slave\_transfer\_t::handle**

**6.5.2.3.0.6.2 i2c\_slave\_transfer\_event\_t i2c\_slave\_transfer\_t::event**

**6.5.2.3.0.6.3 uint8\_t i2c\_slave\_transfer\_t::receivedAddress**

7-bits plus R/nW bit0

**6.5.2.3.0.6.4 uint32\_t i2c\_slave\_transfer\_t::eventMask**

**6.5.2.3.0.6.5 size\_t i2c\_slave\_transfer\_t::transferredCount**

**6.5.2.3.0.6.6 status\_t i2c\_slave\_transfer\_t::completionStatus**

Only applies for [kI2C\\_SlaveCompletionEvent](#).

### 6.5.2.4 struct i2c\_slave\_handle

I2C slave handle typedef.

Note

The contents of this structure are private and subject to change.

### Data Fields

- volatile [i2c\\_slave\\_transfer\\_t](#) [transfer](#)  
*I2C slave transfer.*
- volatile bool [isBusy](#)

- *Whether transfer is busy.*  
volatile `i2c_slave_fsm_t` `slaveFsm`  
*slave transfer state machine.*
- `i2c_slave_transfer_callback_t` `callback`  
*Callback function called at transfer event.*
- void \* `userData`  
*Callback parameter passed to callback.*

#### 6.5.2.4.0.7 Field Documentation

6.5.2.4.0.7.1 volatile `i2c_slave_transfer_t` `i2c_slave_handle_t::transfer`

6.5.2.4.0.7.2 volatile bool `i2c_slave_handle_t::isBusy`

6.5.2.4.0.7.3 volatile `i2c_slave_fsm_t` `i2c_slave_handle_t::slaveFsm`

6.5.2.4.0.7.4 `i2c_slave_transfer_callback_t` `i2c_slave_handle_t::callback`

6.5.2.4.0.7.5 void\* `i2c_slave_handle_t::userData`

### 6.5.3 Typedef Documentation

6.5.3.1 typedef void(\* `i2c_slave_transfer_callback_t`)(`I2C_Type` \*`base`, volatile `i2c_slave_transfer_t` \*`transfer`, void \*`userData`)

This callback is used only for the slave non-blocking transfer API. To install a callback, use the `I2C_SlaveSetCallback()` function after you have created a handle.

Parameters

<i>base</i>	Base address for the I2C instance on which the event occurred.
<i>transfer</i>	Pointer to transfer descriptor containing values passed to and/or from the callback.
<i>userData</i>	Arbitrary pointer-sized value passed from the application.

### 6.5.4 Enumeration Type Documentation

6.5.4.1 enum `i2c_slave_flags`

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

***kI2C\_SlavePendingFlag*** The I2C module is waiting for software interaction.

***kI2C\_SlaveNotStretching*** Indicates whether the slave is currently stretching clock (0 = yes, 1 = no).

## I2C Slave Driver

***kI2C\_SlaveSelected*** Indicates whether the slave is selected by an address match.

***kI2C\_SaveDeselected*** Indicates that slave was previously deselected (deselect event took place, w1c).

### 6.5.4.2 enum i2c\_slave\_address\_register\_t

Enumerator

***kI2C\_SlaveAddressRegister0*** Slave Address 0 register.

***kI2C\_SlaveAddressRegister1*** Slave Address 1 register.

***kI2C\_SlaveAddressRegister2*** Slave Address 2 register.

***kI2C\_SlaveAddressRegister3*** Slave Address 3 register.

### 6.5.4.3 enum i2c\_slave\_address\_qual\_mode\_t

Enumerator

***kI2C\_QualModeMask*** The SLVQUAL0 field (qualAddress) is used as a logical mask for matching address0.

***kI2C\_QualModeExtend*** The SLVQUAL0 (qualAddress) field is used to extend address 0 matching in a range of addresses.

### 6.5.4.4 enum i2c\_slave\_bus\_speed\_t

### 6.5.4.5 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\(\)](#) in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note

These enumerations are meant to be OR'd together to form a bit mask of events.

Enumerator

***kI2C\_SlaveAddressMatchEvent*** Received the slave address after a start or repeated start.

***kI2C\_SlaveTransmitEvent*** Callback is requested to provide data to transmit (slave-transmitter role).

***kI2C\_SlaveReceiveEvent*** Callback is requested to provide a buffer in which to place received data (slave-receiver role).

***kI2C\_SlaveCompletionEvent*** All data in the active transfer have been consumed.

***kI2C\_SlaveDeselectedEvent*** The slave function has become deselected (SLVSEL flag changing from 1 to 0).

***kI2C\_SlaveAllEvents*** Bit mask of all available events.



## 6.5.5 Function Documentation

### 6.5.5.1 void I2C\_SlaveGetDefaultConfig ( i2c\_slave\_config\_t \* *slaveConfig* )

This function provides the following default configuration for the I2C slave peripheral:

```
* slaveConfig->enableSlave = true;
* slaveConfig->address0.disable = false;
* slaveConfig->address0.address = 0u;
* slaveConfig->address1.disable = true;
* slaveConfig->address2.disable = true;
* slaveConfig->address3.disable = true;
* slaveConfig->busSpeed = kI2C_SlaveStandardMode;
*
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with [I2C\\_SlaveInit\(\)](#). Be sure to override at least the *address0.address* member of the configuration structure with the desired slave address.

Parameters

out	<i>slaveConfig</i>	User provided configuration structure that is set to default values. Refer to <a href="#">i2c_slave_config_t</a> .
-----	--------------------	--

### 6.5.5.2 status\_t I2C\_SlaveInit ( I2C\_Type \* *base*, const i2c\_slave\_config\_t \* *slaveConfig*, uint32\_t *srcClock\_Hz* )

This function enables the peripheral clock and initializes the I2C slave peripheral as described by the user provided configuration.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>slaveConfig</i>	User provided peripheral configuration. Use <a href="#">I2C_SlaveGetDefaultConfig()</a> to get a set of defaults that you can override.
<i>srcClock_Hz</i>	Frequency in Hertz of the I2C functional clock. Used to calculate CLKDIV value to provide enough data setup time for master when slave stretches the clock.

### 6.5.5.3 void I2C\_SlaveSetAddress ( I2C\_Type \* *base*, i2c\_slave\_address\_register\_t *addressRegister*, uint8\_t *address*, bool *addressDisable* )

This function writes new value to Slave Address register.

## I2C Slave Driver

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>address-Register</i>	The module supports multiple address registers. The parameter determines which one shall be changed.
<i>address</i>	The slave address to be stored to the address register for matching.
<i>addressDisable</i>	Disable matching of the specified address register.

#### 6.5.5.4 void I2C\_SlaveDeinit ( I2C\_Type \* *base* )

This function disables the I2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

### Parameters

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

#### 6.5.5.5 static void I2C\_SlaveEnable ( I2C\_Type \* *base*, bool *enable* ) [inline], [static]

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>enable</i>	True to enable or false to disable.

#### 6.5.5.6 static void I2C\_SlaveClearStatusFlags ( I2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>statusMask</i>	A bitmask of status flags that are to be cleared. The mask is composed of <a href="#">_i2c_slave_flags</a> enumerators OR'd together. You may pass the result of a previous call to <code>I2C_SlaveGetStatusFlags()</code> .

See Also

[\\_i2c\\_slave\\_flags](#).

#### 6.5.5.7 **status\_t I2C\_SlaveWriteBlocking ( I2C\_Type \* *base*, const uint8\_t \* *txBuff*, size\_t *txSize* )**

The function executes blocking address phase and blocking data phase.

Parameters

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

Returns

kStatus\_Success Data has been sent.

kStatus\_Fail Unexpected slave state (master data write while master read from slave is expected).

#### 6.5.5.8 **status\_t I2C\_SlaveReadBlocking ( I2C\_Type \* *base*, uint8\_t \* *rxBuff*, size\_t *rxSize* )**

The function executes blocking address phase and blocking data phase.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>rxBuff</i>	The pointer to the data to be transferred.
<i>rxSize</i>	The length in bytes of the data to be transferred.

Returns

kStatus\_Success Data has been received.

kStatus\_Fail Unexpected slave state (master data read while master write to slave is expected).

## I2C Slave Driver

### 6.5.5.9 void I2C\_SlaveTransferCreateHandle ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, i2c\_slave\_transfer\_callback\_t *callback*, void \* *userData* )

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the [I2C\\_SlaveTransferAbort\(\)](#) API shall be called.

Parameters

	<i>base</i>	The I2C peripheral base address.
out	<i>handle</i>	Pointer to the I2C slave driver handle.
	<i>callback</i>	User provided pointer to the asynchronous callback function.
	<i>userData</i>	User provided pointer to the application callback data.

### 6.5.5.10 status\_t I2C\_SlaveTransferNonBlocking ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, uint32\_t *eventMask* )

Call this API after calling [I2C\\_SlaveInit\(\)](#) and [I2C\\_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to [I2C\\_SlaveTransferCreateHandle\(\)](#). The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes [kI2C\\_SlaveTransmitEvent](#) callback. If no slave Rx transfer is busy, a master write to slave request invokes [kI2C\\_SlaveReceiveEvent](#) callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The [kI2C\\_SlaveTransmitEvent](#) and [kI2C\\_SlaveReceiveEvent](#) events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [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.

Return values

<i>kStatus_Success</i>	Slave transfers were successfully started.
<i>kStatus_I2C_Busy</i>	Slave transfers have already been started on this handle.

#### 6.5.5.11 **status\_t I2C\_SlaveSetSendBuffer ( I2C\_Type \* *base*, volatile i2c\_slave\_transfer\_t \* *transfer*, const void \* *txData*, size\_t *txSize*, uint32\_t *eventMask* )**

The function can be called in response to [kI2C\\_SlaveTransmitEvent](#) callback to start a new slave Tx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The [kI2C\\_SlaveTransmitEvent](#) and [kI2C\\_SlaveReceiveEvent](#) events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [kI2C\\_SlaveAllEvents](#) constant is provided as a convenient way to enable all events.

Parameters

<i>base</i>	The I2C peripheral base address.
<i>transfer</i>	Pointer to <a href="#">i2c_slave_transfer_t</a> structure.
<i>txData</i>	Pointer to data to send to master.
<i>txSize</i>	Size of <i>txData</i> in bytes.
<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.

Return values

<i>kStatus_Success</i>	Slave transfers were successfully started.
<i>kStatus_I2C_Busy</i>	Slave transfers have already been started on this handle.

#### 6.5.5.12 **status\_t I2C\_SlaveSetReceiveBuffer ( I2C\_Type \* *base*, volatile i2c\_slave\_transfer\_t \* *transfer*, void \* *rxData*, size\_t *rxSize*, uint32\_t *eventMask* )**

The function can be called in response to [kI2C\\_SlaveReceiveEvent](#) callback to start a new slave Rx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the

## I2C Slave Driver

OR'd combination of [i2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The [kI2C\\_SlaveTransmitEvent](#) and [kI2C\\_SlaveReceiveEvent](#) events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [kI2C\\_SlaveAllEvents](#) constant is provided as a convenient way to enable all events.

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>transfer</i>	Pointer to <a href="#">i2c_slave_transfer_t</a> structure.
<i>rxData</i>	Pointer to data to store data from master.
<i>rxSize</i>	Size of rxData in bytes.
<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.

### Return values

<i>kStatus_Success</i>	Slave transfers were successfully started.
<i>kStatus_I2C_Busy</i>	Slave transfers have already been started on this handle.

#### 6.5.5.13 **static uint32\_t I2C\_SlaveGetReceivedAddress ( I2C\_Type \* *base*, volatile i2c\_slave\_transfer\_t \* *transfer* ) [inline], [static]**

This function should only be called from the address match event callback [kI2C\\_SlaveAddressMatchEvent](#).

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>transfer</i>	The I2C slave transfer.

### Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

#### 6.5.5.14 **void I2C\_SlaveTransferAbort ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle* )**

## Note

This API could be called at any time to stop slave for handling the bus events.

## Parameters

<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to i2c_slave_handle_t structure which stores the transfer state.

## Return values

<i>kStatus_Success</i>	
<i>kStatus_I2C_Idle</i>	

#### 6.5.5.15 status\_t I2C\_SlaveTransferGetCount ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, size\_t \* *count* )

## Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_slave_handle_t structure.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

## Return values

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

#### 6.5.5.16 void I2C\_SlaveTransferHandleIRQ ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle* )

## Note

This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

## I2C Slave Driver

### Parameters

<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to i2c_slave_handle_t structure which stores the transfer state.



## 6.6 I2C DMA Driver

### 6.6.1 Overview

#### Data Structures

- struct [i2c\\_master\\_dma\\_handle\\_t](#)  
*I2C master dma transfer structure. [More...](#)*

#### Macros

- #define [I2C\\_MAX\\_DMA\\_TRANSFER\\_COUNT](#) 1024  
*Maximum length of single DMA transfer (determined by capability of the DMA engine)*

#### Typedefs

- typedef void(\* [i2c\\_master\\_dma\\_transfer\\_callback\\_t](#) )(I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, [status\\_t](#) status, void \*userData)  
*I2C master dma transfer callback typedef.*

### I2C Block DMA Transfer Operation

- void [I2C\\_MasterTransferCreateHandleDMA](#) (I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, [i2c\\_master\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, [dma\\_handle\\_t](#) \*dmaHandle)  
*Init 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)  
*Get master transfer status during a dma non-blocking transfer.*
- void [I2C\\_MasterTransferAbortDMA](#) (I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle)  
*Abort a master dma non-blocking transfer in a early time.*

### 6.6.2 Data Structure Documentation

#### 6.6.2.1 struct\_i2c\_master\_dma\_handle

I2C master dma handle typedef.

#### Data Fields

- uint8\_t [state](#)

## I2C DMA Driver

- *Transfer state machine current state.*  
uint32\_t **transferCount**
- *Indicates progress of the transfer.*  
uint32\_t **remainingBytesDMA**
- *Remaining byte count to be transferred using DMA.*  
uint8\_t \* **buf**
- *Buffer pointer for current state.*  
dma\_handle\_t \* **dmaHandle**
- *The DMA handler used.*  
i2c\_master\_transfer\_t **transfer**
- *Copy of the current transfer info.*  
i2c\_master\_dma\_transfer\_callback\_t **completionCallback**
- *Callback function called after dma transfer finished.*  
void \* **userData**
- *Callback parameter passed to callback function.*

### 6.6.2.1.0.8 Field Documentation

6.6.2.1.0.8.1 uint8\_t i2c\_master\_dma\_handle\_t::state

6.6.2.1.0.8.2 uint32\_t i2c\_master\_dma\_handle\_t::remainingBytesDMA

6.6.2.1.0.8.3 uint8\_t\* i2c\_master\_dma\_handle\_t::buf

6.6.2.1.0.8.4 dma\_handle\_t\* i2c\_master\_dma\_handle\_t::dmaHandle

6.6.2.1.0.8.5 i2c\_master\_transfer\_t i2c\_master\_dma\_handle\_t::transfer

6.6.2.1.0.8.6 i2c\_master\_dma\_transfer\_callback\_t i2c\_master\_dma\_handle\_t::completion-  
Callback

6.6.2.1.0.8.7 void\* i2c\_master\_dma\_handle\_t::userData

### 6.6.3 Typedef Documentation

6.6.3.1 typedef void(\* i2c\_master\_dma\_transfer\_callback\_t)(I2C\_Type \*base,  
i2c\_master\_dma\_handle\_t \*handle, status\_t status, void \*userData)

### 6.6.4 Function Documentation

6.6.4.1 void I2C\_MasterTransferCreateHandleDMA ( I2C\_Type \* *base*,  
i2c\_master\_dma\_handle\_t \* *handle*, i2c\_master\_dma\_transfer\_callback\_t  
*callback*, void \* *userData*, dma\_handle\_t \* *dmaHandle* )

## Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	pointer to i2c_master_dma_handle_t structure
<i>callback</i>	pointer to user callback function
<i>userData</i>	user param passed to the callback function
<i>dmaHandle</i>	DMA handle pointer

#### 6.6.4.2 status\_t I2C\_MasterTransferDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle*, i2c\_master\_transfer\_t \* *xfer* )

## Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	pointer to i2c_master_dma_handle_t structure
<i>xfer</i>	pointer to transfer structure of i2c_master_transfer_t

## 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.

#### 6.6.4.3 status\_t I2C\_MasterTransferGetCountDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle*, size\_t \* *count* )

## Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	pointer to i2c_master_dma_handle_t structure

## I2C DMA Driver

<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.
--------------	---

**6.6.4.4** void I2C\_MasterTransferAbortDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle* )

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	pointer to i2c_master_dma_handle_t structure

## 6.7 I2C FreeRTOS Driver

### 6.7.1 Overview

#### Data Structures

- struct [i2c\\_rtos\\_handle\\_t](#)  
*I2C FreeRTOS handle. [More...](#)*

#### 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 I2C transfer.*

### 6.7.2 Data Structure Documentation

#### 6.7.2.1 struct i2c\_rtos\_handle\_t

##### Data Fields

- [I2C\\_Type](#) \* [base](#)  
*I2C base address.*
- [i2c\\_master\\_handle\\_t](#) [drv\\_handle](#)  
*A handle of the underlying driver, treated as opaque by the RTOS layer.*
- [status\\_t](#) [async\\_status](#)  
*Transactional state of the underlying driver.*
- [SemaphoreHandle\\_t](#) [mutex](#)  
*A mutex to lock the handle during a transfer.*
- [SemaphoreHandle\\_t](#) [semaphore](#)  
*A semaphore to notify and unblock task when the transfer ends.*

### 6.7.3 Function Documentation

#### 6.7.3.1 [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.

## I2C FreeRTOS Driver

### 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>	Configuration structure to set-up I2C in master mode.
<i>srcClock_Hz</i>	Frequency of input clock of the I2C module.

### Returns

status of the operation.

#### 6.7.3.2 **status\_t I2C\_RTOS\_Deinit ( i2c\_rtos\_handle\_t \* *handle* )**

This function deinitializes the I2C module and the related RTOS context.

### Parameters

<i>handle</i>	The RTOS I2C handle.
---------------	----------------------

#### 6.7.3.3 **status\_t I2C\_RTOS\_Transfer ( i2c\_rtos\_handle\_t \* *handle*, i2c\_master\_transfer\_t \* *transfer* )**

This function performs an I2C transfer according to data given in the transfer structure.

### Parameters

<i>handle</i>	The RTOS I2C handle.
<i>transfer</i>	Structure specifying the transfer parameters.

### Returns

status of the operation.

## Chapter 7

### I2S: I2S Driver

#### 7.1 Overview

The MCUXpresso SDK provides the peripheral driver for the I2S function of FLEXCOMM module of MCUXpresso SDK devices.

The I2S module is used to transmit or receive digital audio data. Only transmit or receive is enabled at one time in one module.

Driver currently supports one (primary) channel pair per one I2S enabled FLEXCOMM module only.

#### 7.2 I2S Driver Initialization and Configuration

[I2S\\_TxInit\(\)](#) and [I2S\\_RxInit\(\)](#) functions ungate the clock for the FLEXCOMM module, assign I2S function to FLEXCOMM module and configure audio data format and other I2S operational settings. [I2S\\_TxInit\(\)](#) is used when I2S should transmit data, [I2S\\_RxInit\(\)](#) when it should receive data.

[I2S\\_TxGetDefaultConfig\(\)](#) and [I2S\\_RxGetDefaultConfig\(\)](#) functions can be used to set the module configuration structure with default values for transmit and receive function, respectively.

[I2S\\_Deinit\(\)](#) function resets the FLEXCOMM module.

[I2S\\_TxTransferCreateHandle\(\)](#) function creates transactional handle for transmit in interrupt mode.

[I2S\\_RxTransferCreateHandle\(\)](#) function creates transactional handle for receive in interrupt mode.

[I2S\\_TxTransferCreateHandleDMA\(\)](#) function creates transactional handle for transmit in DMA mode.

[I2S\\_RxTransferCreateHandleDMA\(\)](#) function creates transactional handle for receive in DMA mode.

#### 7.3 I2S Transmit Data

[I2S\\_TxTransferNonBlocking\(\)](#) function is used to add data buffer to transmit in interrupt mode. It also begins transmission if not transmitting yet.

[I2S\\_RxTransferNonBlocking\(\)](#) function is used to add data buffer to receive data into in interrupt mode. It also begins reception if not receiving yet.

[I2S\\_TxTransferSendDMA\(\)](#) function is used to add data buffer to transmit in DMA mode. It also begins transmission if not transmitting yet.

[I2S\\_RxTransferReceiveDMA\(\)](#) function is used to add data buffer to receive data into in DMA mode. It also begins reception if not receiving yet.

The transfer of data will be stopped automatically when all data buffers queued using the above functions will be processed and no new data buffer is enqueued meanwhile. If the above functions are not called frequently enough, I2S stop followed by restart may keep occurring resulting in drops audio stream.





Lnn - left channel bit nn

Note that for example if `i2s_config_t.dataLength` = 7, bits on positions R07-R15 and L07-L15 are ignored (buffer "wastes space").

If `i2s_config_t.dataLength` (channel bit width) is between 17 and 24 and `i2s_config_t.pack48` = false:

Even words (counting from zero):

[illegible]

Odd words (counting from zero):

<b>MSB</b>																															<b>LSB</b>
							R2	B2	R2	R2	R2	R1	R1	B1	R1	G1	B1	A1	B1	R1	R1	R1	R0	R0	R0	R0	G0	B0	R0	R0	R0

If `i2s_config_t.dataLength` (channel bit width) is between 17 and 24 and `i2s_config_t.pack48 = true`:

Even words (counting from zero):

MSB																														LSB
R07	R06	R05	R04	R03	R02	R01	Q2	B2	Z2	L2	O1	E1	S1	T1	M1	A1	I1	N1	L1	O0	P0	H0	F0	G0	J0	K0	C0	D0	L00	

Odd words (counting from zero):

[illegible]

If `i2s_config_t.dataLength` (channel bit width) is between 25 and 32:

Even words (counting from zero):

MSB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					
-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Odd words (counting from zero):

MSB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					</
-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

### 7.6.2 Interrupt mode

Buffer does not need to be aligned (buffer is read / written by single bytes, each byte contain left and right channel):

MSB							LSB
R03	R02	R01	R00	L03	L02	L01	L00

Length of buffer for transmit or receive has to be multiply of 2 bytes. Buffer address has to be aligned to 2-bytes.

<b>MSB</b>															<b>LSB</b>
R07	R06	R05	R04	R03	R02	R01	R00	L07	L06	L05	L04	L03	L02	L01	L00

Length of buffer for transmit or receive has to be multiply of 4 bytes. Buffer address has to be aligned to 4-bytes.

[illegible]

Length of buffer for transmit or receive has to be multiply of 6 bytes.

[illegible]

Length of buffer for transmit or receive has to be multiply of 6 bytes. Buffer address has to be aligned to 4-bytes.

[illegible]

If `i2s_config_t.dataLength` (channel bit width) is between 25 and 32 and `i2s_config_t.oneChannel` = false:  
Buffer for transmit or receive has to be multiply of 8 bytes. Buffer address has to be aligned to 4-bytes.

Even words (counting from zero):

<b>MSB</b>																														<b>LSB</b>
L31	L30	Q29	Q28	Z27	Z26	E25	D24	B23	C22	L21	O20	I19	S18	T17	F16	V15	N14	H13	K12	J11	L10	P09	M08	O07	G06	S05	A04	O03	O02	O00

Odd words (counting from zero):

MSB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					
-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

If `i2s_config_t.dataLength` (channel bit width) is between 25 and 32 and `i2s_config_t.oneChannel` = true:

Buffer for transmit or receive has to be multiply of 4 bytes. Buffer address has to be aligned to 4-bytes.

MSB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																				
-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

## 7.7 I2S Driver Examples

### 7.7.1 Interrupt mode examples

## Transmit example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_handle_t handle;

    I2S_TxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_TxInit(I2S0, &config);

    I2S_TxTransferCreateHandle(I2S0, &handle, TxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
    I2S_TxTransferNonBlocking(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_TxTransferNonBlocking(I2S0, &handle, someTransfer);
}

void TxCallback(I2S_Type *base, i2s_handle_t *handle, status_t completionStatus, void *userData)
{
    i2s_transfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
    }
}
```

## I2S Driver Examples

```
        transfer.dataSize = sizeof(buffer);
        I2S_TxTransferNonBlocking(base, handle, transfer);
    }
}
```

### Receive example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_handle_t handle;

    I2S_RxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_RxInit(I2S0, &config);

    I2S_RxTransferCreateHandle(I2S0, &handle, RxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
    I2S_RxTransferNonBlocking(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_RxTransferNonBlocking(I2S0, &handle, someTransfer);
}

void RxCallback(I2S_Type *base, i2s_handle_t *handle, status_t completionStatus, void *userData)
{
    i2s_transfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
        transfer.dataSize = sizeof(buffer);
        I2S_RxTransferNonBlocking(base, handle, transfer);
    }
}
```

## 7.7.2 DMA mode examples

### Transmit example

```
void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_dma_handle_t handle;

    I2S_TxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_TxInit(I2S0, &config);

    I2S_TxTransferCreateHandleDMA(I2S0, &handle, TxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
```

```

I2S_TxTransferNonBlockingDMA(I2S0, &handle, transfer);

/* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
   finishes */
I2S_TxTransferNonBlockingDMA(I2S0, &handle, someTransfer);
}

void TxCallback(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData)
{
    i2s_transfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
        transfer.dataSize = sizeof(buffer);
        I2S_TxTransferNonBlockingDMA(base, handle, transfer);
    }
}

```

## Receive example

```

void StartTransfer(void)
{
    i2s_config_t config;
    i2s_transfer_t transfer;
    i2s_dma_handle_t handle;

    I2S_RxGetDefaultConfig(&config);
    config.masterSlave = kI2S_MasterSlaveNormalMaster;
    config.divider = 32; /* clock frequency/audio sample frequency/channels/channel bit depth */
    I2S_RxInit(I2S0, &config);

    I2S_RxTransferCreateHandleDMA(I2S0, &handle, RxCallback, NULL);

    transfer.data = buffer;
    transfer.dataSize = sizeof(buffer);
    I2S_RxTransferNonBlockingDMA(I2S0, &handle, transfer);

    /* Enqueue next buffer right away so there is no drop in audio data stream when the first buffer
       finishes */
    I2S_RxTransferNonBlockingDMA(I2S0, &handle, someTransfer);
}

void RxCallback(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData)
{
    i2s_transfer_t transfer;

    if (completionStatus == kStatus_I2S_BufferComplete)
    {
        /* Enqueue next buffer */
        transfer.data = buffer;
        transfer.dataSize = sizeof(buffer);
        I2S_RxTransferNonBlockingDMA(base, handle, transfer);
    }
}

```

## Modules

- [I2S DMA Driver](#)
- [I2S Driver](#)

## I2S Driver

### 7.8 I2S Driver

#### 7.8.1 Overview

#### Files

- file [fsl\\_i2s.h](#)

#### Data Structures

- struct [i2s\\_config\\_t](#)  
*I2S configuration structure. [More...](#)*
- struct [i2s\\_transfer\\_t](#)  
*Buffer to transfer from or receive audio data into. [More...](#)*
- struct [i2s\\_handle\\_t](#)  
*Members not to be accessed / modified outside of the driver. [More...](#)*

#### Macros

- #define [I2S\\_NUM\\_BUFFERS](#) (4)  
*Number of buffers .*

#### Typedefs

- typedef void(\* [i2s\\_transfer\\_callback\\_t](#) )(I2S\_Type \*base, i2s\_handle\_t \*handle, [status\\_t](#) completionStatus, void \*userData)  
*Callback function invoked from transactional API on completion of a single buffer transfer.*

#### Enumerations

- enum [\\_i2s\\_status](#) {  
    [kStatus\\_I2S\\_BufferComplete](#),  
    [kStatus\\_I2S\\_Done](#) = MAKE\_STATUS(kStatusGroup\_I2S, 1),  
    [kStatus\\_I2S\\_Busy](#) }  
*I2S status codes.*
- enum [i2s\\_flags\\_t](#) {  
    [kI2S\\_TxErrorFlag](#) = I2S\_FIFOINTENSET\_TXERR\_MASK,  
    [kI2S\\_TxLevelFlag](#) = I2S\_FIFOINTENSET\_TXLVL\_MASK,  
    [kI2S\\_RxErrorFlag](#) = I2S\_FIFOINTENSET\_RXERR\_MASK,  
    [kI2S\\_RxLevelFlag](#) = I2S\_FIFOINTENSET\_RXLVL\_MASK }  
*I2S flags.*

- enum `i2s_master_slave_t` {  
`kI2S_MasterSlaveNormalSlave` = 0x0,  
`kI2S_MasterSlaveWsSyncMaster` = 0x1,  
`kI2S_MasterSlaveExtSckMaster` = 0x2,  
`kI2S_MasterSlaveNormalMaster` = 0x3 }  
*Master / slave mode.*
- enum `i2s_mode_t` {  
`kI2S_ModeI2sClassic` = 0x0,  
`kI2S_ModeDspWs50` = 0x1,  
`kI2S_ModeDspWsShort` = 0x2,  
`kI2S_ModeDspWsLong` = 0x3 }  
*I2S mode.*

## Driver version

- #define `FSL_I2S_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*I2S driver version 2.0.0.*

## Initialization and deinitialization

- void `I2S_TxInit` (`I2S_Type *base`, const `i2s_config_t *config`)  
*Initializes the FLEXCOMM peripheral for I2S transmit functionality.*
- void `I2S_RxInit` (`I2S_Type *base`, const `i2s_config_t *config`)  
*Initializes the FLEXCOMM peripheral for I2S receive functionality.*
- void `I2S_TxGetDefaultConfig` (`i2s_config_t *config`)  
*Sets the I2S Tx configuration structure to default values.*
- void `I2S_RxGetDefaultConfig` (`i2s_config_t *config`)  
*Sets the I2S Rx configuration structure to default values.*
- void `I2S_Deinit` (`I2S_Type *base`)  
*De-initializes the I2S peripheral.*

## Non-blocking API

- void `I2S_TxTransferCreateHandle` (`I2S_Type *base`, `i2s_handle_t *handle`, `i2s_transfer_callback_t` callback, void \*userData)  
*Initializes handle for transfer of audio data.*
- `status_t` `I2S_TxTransferNonBlocking` (`I2S_Type *base`, `i2s_handle_t *handle`, `i2s_transfer_t` transfer)  
*Begins or queue sending of the given data.*
- void `I2S_TxTransferAbort` (`I2S_Type *base`, `i2s_handle_t *handle`)  
*Aborts sending of data.*
- void `I2S_RxTransferCreateHandle` (`I2S_Type *base`, `i2s_handle_t *handle`, `i2s_transfer_callback_t` callback, void \*userData)  
*Initializes handle for reception of audio data.*

## I2S Driver

- [status\\_t I2S\\_RxTransferNonBlocking](#) (I2S\_Type \*base, i2s\_handle\_t \*handle, [i2s\\_transfer\\_t](#) transfer)  
*Begins or queue reception of data into given buffer.*
- void [I2S\\_RxTransferAbort](#) (I2S\_Type \*base, i2s\_handle\_t \*handle)  
*Aborts receiving of data.*
- [status\\_t I2S\\_TransferGetCount](#) (I2S\_Type \*base, i2s\_handle\_t \*handle, size\_t \*count)  
*Returns number of bytes transferred so far.*
- [status\\_t I2S\\_TransferGetErrorCount](#) (I2S\_Type \*base, i2s\_handle\_t \*handle, size\_t \*count)  
*Returns number of buffer underruns or overruns.*

## Enable / disable

- static void [I2S\\_Enable](#) (I2S\_Type \*base)  
*Enables I2S operation.*
- static void [I2S\\_Disable](#) (I2S\_Type \*base)  
*Disables I2S operation.*

## Interrupts

- static void [I2S\\_EnableInterrupts](#) (I2S\_Type \*base, uint32\_t interruptMask)  
*Enables I2S FIFO interrupts.*
- static void [I2S\\_DisableInterrupts](#) (I2S\_Type \*base, uint32\_t interruptMask)  
*Disables I2S FIFO interrupts.*
- static uint32\_t [I2S\\_GetEnabledInterrupts](#) (I2S\_Type \*base)  
*Returns the set of currently enabled I2S FIFO interrupts.*
- void [I2S\\_TxHandleIRQ](#) (I2S\_Type \*base, i2s\_handle\_t \*handle)  
*Invoked from interrupt handler when transmit FIFO level decreases.*
- void [I2S\\_RxHandleIRQ](#) (I2S\_Type \*base, i2s\_handle\_t \*handle)  
*Invoked from interrupt handler when receive FIFO level decreases.*

## 7.8.2 Data Structure Documentation

### 7.8.2.1 struct i2s\_config\_t

#### Data Fields

- [i2s\\_master\\_slave\\_t](#) masterSlave  
*Master / slave configuration.*
- [i2s\\_mode\\_t](#) mode  
*I2S mode.*
- bool [rightLow](#)  
*Right channel data in low portion of FIFO.*
- bool [leftJust](#)  
*Left justify data in FIFO.*
- bool [sckPol](#)  
*SCK polarity.*



- bool [wsPol](#)  
*WS polarity.*
- uint16\_t [divider](#)  
*Flexcomm function clock divider (1 - 4096)*
- bool [oneChannel](#)  
*true mono, false stereo*
- uint8\_t [dataLength](#)  
*Data length (4 - 32)*
- uint16\_t [frameLength](#)  
*Frame width (4 - 512)*
- uint16\_t [position](#)  
*Data position in the frame.*
- uint8\_t [watermark](#)  
*FIFO trigger level.*
- bool [txEmptyZero](#)  
*Transmit zero when buffer becomes empty or last item.*
- bool [pack48](#)  
*Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)*

### 7.8.2.2 struct i2s\_transfer\_t

#### Data Fields

- volatile uint8\_t \* [data](#)  
*Pointer to data buffer.*
- volatile size\_t [dataSize](#)  
*Buffer size in bytes.*

#### 7.8.2.2.0.9 Field Documentation

##### 7.8.2.2.0.9.1 volatile uint8\_t\* i2s\_transfer\_t::data

##### 7.8.2.2.0.9.2 volatile size\_t i2s\_transfer\_t::dataSize

### 7.8.2.3 struct \_i2s\_handle

Transactional state of the initialized transfer or receive I2S operation.

#### Data Fields

- uint32\_t [state](#)  
*State of transfer.*
- [i2s\\_transfer\\_callback\\_t](#) [completionCallback](#)  
*Callback function pointer.*
- void \* [userData](#)  
*Application data passed to callback.*
- bool [oneChannel](#)  
*true mono, false stereo*
- uint8\_t [dataLength](#)

## I2S Driver

- *Data length (4 - 32)*  
• bool [pack48](#)  
*Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)*
- bool [useFifo48H](#)  
*When dataLength 17-24: true use FIFOWR48H, false use FIFOWR.*
- volatile [i2s\\_transfer\\_t](#) [i2sQueue](#) [[I2S\\_NUM\\_BUFFERS](#)]  
*Transfer queue storing transfer buffers.*
- volatile uint8\_t [queueUser](#)  
*Queue index where user's next transfer will be stored.*
- volatile uint8\_t [queueDriver](#)  
*Queue index of buffer actually used by the driver.*
- volatile uint32\_t [errorCount](#)  
*Number of buffer underruns/overruns.*
- volatile uint32\_t [transferCount](#)  
*Number of bytes transferred.*
- volatile uint8\_t [watermark](#)  
*FIFO trigger level.*

### 7.8.3 Macro Definition Documentation

#### 7.8.3.1 #define FSL\_I2S\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

Current version: 2.0.0

Change log:

- Version 2.0.0
  - initial version

#### 7.8.3.2 #define I2S\_NUM\_BUFFERS (4)

### 7.8.4 Typedef Documentation

#### 7.8.4.1 typedef void(\* i2s\_transfer\_callback\_t)(I2S\_Type \*base, i2s\_handle\_t \*handle, status\_t completionStatus, void \*userData)

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to I2S transaction.

<i>completion-Status</i>	status of the transaction.
<i>userData</i>	optional pointer to user arguments data.

## 7.8.5 Enumeration Type Documentation

### 7.8.5.1 enum `i2s_status`

Enumerator

***kStatus\_I2S\_BufferComplete*** Transfer from/into a single buffer has completed.

***kStatus\_I2S\_Done*** All buffers transfers have completed.

***kStatus\_I2S\_Busy*** Already performing a transfer and cannot queue another buffer.

### 7.8.5.2 enum `i2s_flags_t`

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

***kI2S\_TxErrorFlag*** TX error interrupt.

***kI2S\_TxLevelFlag*** TX level interrupt.

***kI2S\_RxErrorFlag*** RX error interrupt.

***kI2S\_RxLevelFlag*** RX level interrupt.

### 7.8.5.3 enum `i2s_master_slave_t`

Enumerator

***kI2S\_MasterSlaveNormalSlave*** Normal slave.

***kI2S\_MasterSlaveWsSyncMaster*** WS synchronized master.

***kI2S\_MasterSlaveExtSckMaster*** Master using existing SCK.

***kI2S\_MasterSlaveNormalMaster*** Normal master.

### 7.8.5.4 enum `i2s_mode_t`

Enumerator

***kI2S\_ModeI2sClassic*** I2S classic mode.

***kI2S\_ModeDspWs50*** DSP mode, WS having 50% duty cycle.

***kI2S\_ModeDspWsShort*** DSP mode, WS having one clock long pulse.

***kI2S\_ModeDspWsLong*** DSP mode, WS having one data slot long pulse.

### 7.8.6 Function Documentation

#### 7.8.6.1 void I2S\_TxInit ( I2S\_Type \* *base*, const i2s\_config\_t \* *config* )

Un-gates the FLEXCOMM clock and configures the module for I2S transmission using a configuration structure. The configuration structure can be custom filled or set with default values by [I2S\\_TxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the I2S driver.

Parameters

<i>base</i>	I2S base pointer.
<i>config</i>	pointer to I2S configuration structure.

#### 7.8.6.2 void I2S\_RxInit ( I2S\_Type \* *base*, const i2s\_config\_t \* *config* )

Un-gates the FLEXCOMM clock and configures the module for I2S receive using a configuration structure. The configuration structure can be custom filled or set with default values by [I2S\\_RxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the I2S driver.

Parameters

<i>base</i>	I2S base pointer.
<i>config</i>	pointer to I2S configuration structure.

#### 7.8.6.3 void I2S\_TxGetDefaultConfig ( i2s\_config\_t \* *config* )

This API initializes the configuration structure for use in [I2S\\_TxInit\(\)](#). The initialized structure can remain unchanged in [I2S\\_TxInit\(\)](#), or it can be modified before calling [I2S\\_TxInit\(\)](#). Example:

```
i2s_config_t config;  
I2S_TxGetDefaultConfig(&config);
```

Default values:

```

*  config->masterSlave = kI2S_MasterSlaveNormalMaster;
*  config->mode = kI2S_ModeI2sClassic;
*  config->rightLow = false;
*  config->leftJust = false;
*  config->pdmData = false;
*  config->sckPol = false;
*  config->wsPol = false;
*  config->divider = 1;
*  config->oneChannel = false;
*  config->dataLength = 16;
*  config->frameLength = 32;
*  config->position = 0;
*  config->watermark = 4;
*  config->txEmptyZero = true;
*  config->pack48 = false;
*

```

### Parameters

<i>config</i>	pointer to I2S configuration structure.
---------------	---

#### 7.8.6.4 void I2S\_RxGetDefaultConfig ( i2s\_config\_t \* *config* )

This API initializes the configuration structure for use in [I2S\\_RxInit\(\)](#). The initialized structure can remain unchanged in [I2S\\_RxInit\(\)](#), or it can be modified before calling [I2S\\_RxInit\(\)](#). Example:

```

i2s_config_t config;
I2S_RxGetDefaultConfig(&config);

```

### Default values:

```

*  config->masterSlave = kI2S_MasterSlaveNormalSlave;
*  config->mode = kI2S_ModeI2sClassic;
*  config->rightLow = false;
*  config->leftJust = false;
*  config->pdmData = false;
*  config->sckPol = false;
*  config->wsPol = false;
*  config->divider = 1;
*  config->oneChannel = false;
*  config->dataLength = 16;
*  config->frameLength = 32;
*  config->position = 0;
*  config->watermark = 4;
*  config->txEmptyZero = false;
*  config->pack48 = false;
*

```

## I2S Driver

### Parameters

<i>config</i>	pointer to I2S configuration structure.
---------------	---

### 7.8.6.5 void I2S\_Deinit ( I2S\_Type \* *base* )

This API gates the FLEXCOMM clock. The I2S module can't operate unless I2S\_TxInit or I2S\_RxInit is called to enable the clock.

### Parameters

<i>base</i>	I2S base pointer.
-------------	-------------------

### 7.8.6.6 void I2S\_TxTransferCreateHandle ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle*, i2s\_transfer\_callback\_t *callback*, void \* *userData* )

### Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>callback</i>	function to be called back when transfer is done or fails.
<i>userData</i>	pointer to data passed to callback.

### 7.8.6.7 status\_t I2S\_TxTransferNonBlocking ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle*, i2s\_transfer\_t *transfer* )

### Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>transfer</i>	data buffer.

### Return values

---

<i>kStatus_Success</i>	
<i>kStatus_I2S_Busy</i>	if all queue slots are occupied with unsent buffers.

#### 7.8.6.8 void I2S\_TxTransferAbort ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.

#### 7.8.6.9 void I2S\_RxTransferCreateHandle ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle*, i2s\_transfer\_callback\_t *callback*, void \* *userData* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>callback</i>	function to be called back when transfer is done or fails.
<i>userData</i>	pointer to data passed to callback.

#### 7.8.6.10 status\_t I2S\_RxTransferNonBlocking ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle*, i2s\_transfer\_t *transfer* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>transfer</i>	data buffer.

Return values

<i>kStatus_Success</i>	
------------------------	--

## I2S Driver

<i>kStatus_I2S_Busy</i>	if all queue slots are occupied with buffers which are not full.
-------------------------	--

### 7.8.6.11 void I2S\_RxTransferAbort ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.

### 7.8.6.12 status\_t I2S\_TransferGetCount ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

	<i>base</i>	I2S base pointer.
	<i>handle</i>	pointer to handle structure.
out	<i>count</i>	number of bytes transferred so far by the non-blocking transaction.

Return values

<i>kStatus_Success</i>	
<i>kStatus_NoTransferInProgress</i>	there is no non-blocking transaction currently in progress.

### 7.8.6.13 status\_t I2S\_TransferGetErrorCount ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

	<i>base</i>	I2S base pointer.
	<i>handle</i>	pointer to handle structure.
out	<i>count</i>	number of transmit errors encountered so far by the non-blocking transaction.



## Return values

<i>kStatus_Success</i>	
<i>kStatus_NoTransferInProgress</i>	there is no non-blocking transaction currently in progress.

**7.8.6.14 static void I2S\_Enable ( I2S\_Type \* *base* ) [inline], [static]**

## Parameters

<i>base</i>	I2S base pointer.
-------------	-------------------

**7.8.6.15 static void I2S\_Disable ( I2S\_Type \* *base* ) [inline], [static]**

## Parameters

<i>base</i>	I2S base pointer.
-------------	-------------------

**7.8.6.16 static void I2S\_EnableInterrupts ( I2S\_Type \* *base*, uint32\_t *interruptMask* ) [inline], [static]**

## Parameters

<i>base</i>	I2S base pointer.
<i>interruptMask</i>	bit mask of interrupts to enable. See <a href="#">i2s_flags_t</a> for the set of constants that should be OR'd together to form the bit mask.

**7.8.6.17 static void I2S\_DisableInterrupts ( I2S\_Type \* *base*, uint32\_t *interruptMask* ) [inline], [static]**

## Parameters

<i>base</i>	I2S base pointer.
-------------	-------------------

## I2S Driver

<i>interruptMask</i>	bit mask of interrupts to enable. See <a href="#">i2s_flags_t</a> for the set of constants that should be OR'd together to form the bit mask.
----------------------	---

### 7.8.6.18 static uint32\_t I2S\_GetEnabledInterrupts ( I2S\_Type \* *base* ) [inline], [static]

#### Parameters

<i>base</i>	I2S base pointer.
-------------	-------------------

#### Returns

A bitmask composed of [i2s\\_flags\\_t](#) enumerators OR'd together to indicate the set of enabled interrupts.

### 7.8.6.19 void I2S\_TxHandleIRQ ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle* )

#### Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.

### 7.8.6.20 void I2S\_RxHandleIRQ ( I2S\_Type \* *base*, i2s\_handle\_t \* *handle* )

#### Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.

## 7.9 I2S DMA Driver

### 7.9.1 Overview

#### Data Structures

- struct [i2s\\_dma\\_handle\\_t](#)  
*Members not to be accessed / modified outside of the driver. [More...](#)*

#### Typedefs

- typedef void(\* [i2s\\_dma\\_transfer\\_callback\\_t](#))(I2S\_Type \*base, i2s\_dma\_handle\_t \*handle, [status\\_t](#) completionStatus, void \*userData)  
*Callback function invoked from DMA API on completion.*

#### DMA API

- void [I2S\\_TxTransferCreateHandleDMA](#) (I2S\_Type \*base, i2s\_dma\_handle\_t \*handle, [dma\\_handle\\_t](#) \*dmaHandle, [i2s\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData)  
*Initializes handle for transfer of audio data.*
- [status\\_t](#) [I2S\\_TxTransferSendDMA](#) (I2S\_Type \*base, i2s\_dma\_handle\_t \*handle, [i2s\\_transfer\\_t](#) transfer)  
*Begins or queue sending of the given data.*
- void [I2S\\_TransferAbortDMA](#) (I2S\_Type \*base, i2s\_dma\_handle\_t \*handle)  
*Aborts transfer of data.*
- void [I2S\\_RxTransferCreateHandleDMA](#) (I2S\_Type \*base, i2s\_dma\_handle\_t \*handle, [dma\\_handle\\_t](#) \*dmaHandle, [i2s\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData)  
*Initializes handle for reception of audio data.*
- [status\\_t](#) [I2S\\_RxTransferReceiveDMA](#) (I2S\_Type \*base, i2s\_dma\_handle\_t \*handle, [i2s\\_transfer\\_t](#) transfer)  
*Begins or queue reception of data into given buffer.*
- void [I2S\\_DMACallback](#) ([dma\\_handle\\_t](#) \*handle, void \*userData, bool transferDone, uint32\_t tcDs)  
*Invoked from DMA interrupt handler.*

### 7.9.2 Data Structure Documentation

#### 7.9.2.1 struct \_i2s\_dma\_handle

##### Data Fields

- uint32\_t [state](#)  
*Internal state of I2S DMA transfer.*
- [i2s\\_dma\\_transfer\\_callback\\_t](#) [completionCallback](#)  
*Callback function pointer.*
- void \* [userData](#)

## I2S DMA Driver

- *Application data passed to callback.*  
`dma_handle_t * dmaHandle`  
*DMA handle.*
- volatile `i2s_transfer_t i2sQueue [I2S_NUM_BUFFERS]`  
*Transfer queue storing transfer buffers.*
- volatile `uint8_t queueUser`  
*Queue index where user's next transfer will be stored.*
- volatile `uint8_t queueDriver`  
*Queue index of buffer actually used by the driver.*

### 7.9.3 Typedef Documentation

#### 7.9.3.1 `typedef void(* i2s_dma_transfer_callback_t)(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData)`

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to I2S transaction.
<i>completion-Status</i>	status of the transaction.
<i>userData</i>	optional pointer to user arguments data.

### 7.9.4 Function Documentation

#### 7.9.4.1 `void I2S_TxTransferCreateHandleDMA ( I2S_Type * base, i2s_dma_handle_t * handle, dma_handle_t * dmaHandle, i2s_dma_transfer_callback_t callback, void * userData )`

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>dmaHandle</i>	pointer to dma handle structure.
<i>callback</i>	function to be called back when transfer is done or fails.

<i>userData</i>	pointer to data passed to callback.
-----------------	-------------------------------------

#### 7.9.4.2 status\_t I2S\_TxTransferSendDMA ( I2S\_Type \* *base*, i2s\_dma\_handle\_t \* *handle*, i2s\_transfer\_t *transfer* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>transfer</i>	data buffer.

Return values

<i>kStatus_Success</i>	
<i>kStatus_I2S_Busy</i>	if all queue slots are occupied with unsent buffers.

#### 7.9.4.3 void I2S\_TransferAbortDMA ( I2S\_Type \* *base*, i2s\_dma\_handle\_t \* *handle* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.

#### 7.9.4.4 void I2S\_RxTransferCreateHandleDMA ( I2S\_Type \* *base*, i2s\_dma\_handle\_t \* *handle*, dma\_handle\_t \* *dmaHandle*, i2s\_dma\_transfer\_callback\_t *callback*, void \* *userData* )

Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>dmaHandle</i>	pointer to dma handle structure.

## I2S DMA Driver

<i>callback</i>	function to be called back when transfer is done or fails.
<i>userData</i>	pointer to data passed to callback.

### 7.9.4.5 **status\_t I2S\_RxTransferReceiveDMA ( I2S\_Type \* *base*, i2s\_dma\_handle\_t \* *handle*, i2s\_transfer\_t *transfer* )**

#### Parameters

<i>base</i>	I2S base pointer.
<i>handle</i>	pointer to handle structure.
<i>transfer</i>	data buffer.

#### Return values

<i>kStatus_Success</i>	
<i>kStatus_I2S_Busy</i>	if all queue slots are occupied with buffers which are not full.

### 7.9.4.6 **void I2S\_DMACallback ( dma\_handle\_t \* *handle*, void \* *userData*, bool *transferDone*, uint32\_t *tcds* )**

#### Parameters

<i>handle</i>	pointer to DMA handle structure.
<i>userData</i>	argument for user callback.
<i>transferDone</i>	if transfer was done.
<i>tcds</i>	

## Chapter 8

# SPI: Serial Peripheral Interface Driver

### 8.1 Overview

SPI driver includes functional APIs and transactional APIs.

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

Transactional APIs are transaction target high level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the `spi_handle_t` as the first parameter. Initialize the handle by calling the [SPI\\_MasterTransferCreateHandle\(\)](#) or [SPI\\_SlaveTransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer. This means that the functions [SPI\\_MasterTransferNonBlocking\(\)](#) and [SPI\\_SlaveTransferNonBlocking\(\)](#) set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_SPI_Idle` status.

### 8.2 Typical use case

#### 8.2.1 SPI master transfer using an interrupt method

```
#define BUFFER_LEN (64)
spi_master_handle_t spiHandle;
spi_master_config_t masterConfig;
spi_transfer_t xfer;
volatile bool isFinished = false;

const uint8_t sendData[BUFFER_LEN] = [.....];
uint8_t receiveBuff[BUFFER_LEN];

void SPI_UserCallback(SPI_Type *base, spi_master_handle_t *handle, status_t status, void *userData)
{
    isFinished = true;
}

void main(void)
{
    //...

    SPI_MasterGetDefaultConfig(&masterConfig);

    SPI_MasterInit(SPI0, &masterConfig, srcClock_Hz);
    SPI_MasterTransferCreateHandle(SPI0, &spiHandle, SPI_UserCallback, NULL);

    // Prepare to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
```

## Typical use case

```
xfer.dataSize = sizeof(sendData);

// Send out.
SPI_MasterTransferNonBlocking(SPI0, &spiHandle, &xfer);

// Wait send finished.
while (!isFinished)
{
}

// ...
}
```

### 8.2.2 SPI Send/receive using a DMA method

```
#define BUFFER_LEN (64)
spi_dma_handle_t spiHandle;
dma_handle_t g_spiTxDmaHandle;
dma_handle_t g_spiRxDmaHandle;
spi_config_t masterConfig;
spi_transfer_t xfer;
volatile bool isFinished;

uint8_t sendData[BUFFER_LEN] = ...;
uint8_t receiveBuff[BUFFER_LEN];

void SPI_UserCallback(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)
{
    isFinished = true;
}

void main(void)
{
    //...

    // Initialize DMA peripheral
    DMA_Init(DMA0);

    // Initialize SPI peripheral
    SPI_MasterGetDefaultConfig(&masterConfig);
    masterConfig.sselNum = SPI_SSEL;
    SPI_MasterInit(SPI0, &masterConfig, srcClock_Hz);

    // Enable DMA channels connected to SPI0 Tx/SPI0 Rx request lines
    DMA_EnableChannel(SPI0, SPI_MASTER_TX_CHANNEL);
    DMA_EnableChannel(SPI0, SPI_MASTER_RX_CHANNEL);

    // Set DMA channels priority
    DMA_SetChannelPriority(SPI0, SPI_MASTER_TX_CHANNEL,
        kDMA_ChannelPriority3);
    DMA_SetChannelPriority(SPI0, SPI_MASTER_RX_CHANNEL,
        kDMA_ChannelPriority2);

    // Creates the DMA handle.
    DMA_CreateHandle(&masterTxHandle, SPI0, SPI_MASTER_TX_CHANNEL);
    DMA_CreateHandle(&masterRxHandle, SPI0, SPI_MASTER_RX_CHANNEL);

    // Create SPI DMA handle
    SPI_MasterTransferCreateHandleDMA(SPI0, spiHandle, SPI_UserCallback,
        NULL, &g_spiTxDmaHandle, &g_spiRxDmaHandle);

    // Prepares to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
    xfer.dataSize = sizeof(sendData);
```



```
// Sends out.  
SPI_MasterTransferDMA(SPI0, &spiHandle, &xfer);  
  
// Waits for send to complete.  
while (!isFinished)  
{  
}  
  
// ...  
}
```

## Modules

- [SPI DMA Driver](#)
- [SPI Driver](#)
- [SPI FreeRTOS driver](#)

## SPI Driver

### 8.3 SPI Driver

#### 8.3.1 Overview

This section describes the programming interface of the SPI DMA driver.

#### Files

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

#### Data Structures

- struct [spi\\_delay\\_config\\_t](#)  
*SPI delay time configure structure. [More...](#)*
- struct [spi\\_master\\_config\\_t](#)  
*SPI master user configure structure. [More...](#)*
- struct [spi\\_slave\\_config\\_t](#)  
*SPI slave user configure structure. [More...](#)*
- struct [spi\\_transfer\\_t](#)  
*SPI transfer structure. [More...](#)*
- struct [spi\\_half\\_duplex\\_transfer\\_t](#)  
*SPI half-duplex(master only) transfer structure. [More...](#)*
- struct [spi\\_config\\_t](#)  
*Internal configuration structure used in 'spi' and 'spi\_dma' driver. [More...](#)*
- struct [spi\\_master\\_handle\\_t](#)  
*SPI transfer handle structure. [More...](#)*

#### Macros

- #define [SPI\\_DUMMYDATA](#) (0xFFU)  
*SPI dummy transfer data, the data is sent while txBuff is NULL.*

#### Typedefs

- typedef [spi\\_master\\_handle\\_t](#) [spi\\_slave\\_handle\\_t](#)  
*Slave handle type.*
- typedef void(\* [spi\\_master\\_callback\\_t](#))(SPI\_Type \*base, spi\_master\_handle\_t \*handle, [status\\_t](#) status, void \*userData)  
*SPI master callback for finished transmit.*
- typedef void(\* [spi\\_slave\\_callback\\_t](#))(SPI\_Type \*base, [spi\\_slave\\_handle\\_t](#) \*handle, [status\\_t](#) status, void \*userData)  
*SPI slave callback for finished transmit.*

## Enumerations

- enum `spi_xfer_option_t` {  
`kSPI_FrameDelay` = (SPI\_FIFOWR\_EOF\_MASK),  
`kSPI_FrameAssert` = (SPI\_FIFOWR\_EOT\_MASK) }  
*SPI transfer option.*
- enum `spi_shift_direction_t` {  
`kSPI_MsbFirst` = 0U,  
`kSPI_LsbFirst` = 1U }  
*SPI data shifter direction options.*
- enum `spi_clock_polarity_t` {  
`kSPI_ClockPolarityActiveHigh` = 0x0U,  
`kSPI_ClockPolarityActiveLow` }  
*SPI clock polarity configuration.*
- enum `spi_clock_phase_t` {  
`kSPI_ClockPhaseFirstEdge` = 0x0U,  
`kSPI_ClockPhaseSecondEdge` }  
*SPI clock phase configuration.*
- enum `spi_txfifo_watermark_t` {  
`kSPI_TxFifo0` = 0,  
`kSPI_TxFifo1` = 1,  
`kSPI_TxFifo2` = 2,  
`kSPI_TxFifo3` = 3,  
`kSPI_TxFifo4` = 4,  
`kSPI_TxFifo5` = 5,  
`kSPI_TxFifo6` = 6,  
`kSPI_TxFifo7` = 7 }  
*txFIFO watermark values*
- enum `spi_rxfifo_watermark_t` {  
`kSPI_RxFifo1` = 0,  
`kSPI_RxFifo2` = 1,  
`kSPI_RxFifo3` = 2,  
`kSPI_RxFifo4` = 3,  
`kSPI_RxFifo5` = 4,  
`kSPI_RxFifo6` = 5,  
`kSPI_RxFifo7` = 6,  
`kSPI_RxFifo8` = 7 }  
*rxFIFO watermark values*
- enum `spi_data_width_t` {

## SPI Driver

```
kSPI_Data4Bits = 3,  
kSPI_Data5Bits = 4,  
kSPI_Data6Bits = 5,  
kSPI_Data7Bits = 6,  
kSPI_Data8Bits = 7,  
kSPI_Data9Bits = 8,  
kSPI_Data10Bits = 9,  
kSPI_Data11Bits = 10,  
kSPI_Data12Bits = 11,  
kSPI_Data13Bits = 12,  
kSPI_Data14Bits = 13,  
kSPI_Data15Bits = 14,  
kSPI_Data16Bits = 15 }
```

*Transfer data width.*

- enum `spl_ssel_t` {  
    `kSPI_Ssel0` = 0,  
    `kSPI_Ssel1` = 1,  
    `kSPI_Ssel2` = 2,  
    `kSPI_Ssel3` = 3 }

*Slave select.*

- enum `spl_spol_t`  
    *ssel polarity*
- enum `_spl_status` {  
    `kStatus_SPI_Busy` = MAKE\_STATUS(kStatusGroup\_LPC\_SPI, 0),  
    `kStatus_SPI_Idle` = MAKE\_STATUS(kStatusGroup\_LPC\_SPI, 1),  
    `kStatus_SPI_Error` = MAKE\_STATUS(kStatusGroup\_LPC\_SPI, 2),  
    `kStatus_SPI_BaudrateNotSupport` }

*SPI transfer status.*

- enum `_spl_interrupt_enable` {  
    `kSPI_RxLvlIrq` = SPI\_FIFOINTENSET\_RXLVL\_MASK,  
    `kSPI_TxLvlIrq` = SPI\_FIFOINTENSET\_TXLVL\_MASK }

*SPI interrupt sources.*

- enum `_spl_statusflags` {  
    `kSPI_TxEmptyFlag` = SPI\_FIFOSTAT\_TXEMPTY\_MASK,  
    `kSPI_TxNotFullFlag` = SPI\_FIFOSTAT\_TXNOTFULL\_MASK,  
    `kSPI_RxNotEmptyFlag` = SPI\_FIFOSTAT\_RXNOTEMPTY\_MASK,  
    `kSPI_RxFullFlag` = SPI\_FIFOSTAT\_RXFULL\_MASK }

*SPI status flags.*

## Functions

- uint32\_t `SPI_GetInstance` (SPI\_Type \*base)  
    *Returns instance number for SPI peripheral base address.*

## Driver version

- #define **FSL\_SPI\_DRIVER\_VERSION** (**MAKE\_VERSION**(2, 0, 2))  
*SPI driver version 2.0.2.*

## Initialization and deinitialization

- void **SPI\_MasterGetDefaultConfig** (**spi\_master\_config\_t** \*config)  
*Sets the SPI master configuration structure to default values.*
- **status\_t SPI\_MasterInit** (**SPI\_Type** \*base, const **spi\_master\_config\_t** \*config, **uint32\_t** srcClock\_Hz)  
*Initializes the SPI with master configuration.*
- void **SPI\_SlaveGetDefaultConfig** (**spi\_slave\_config\_t** \*config)  
*Sets the SPI slave configuration structure to default values.*
- **status\_t SPI\_SlaveInit** (**SPI\_Type** \*base, const **spi\_slave\_config\_t** \*config)  
*Initializes the SPI with slave configuration.*
- void **SPI\_Deinit** (**SPI\_Type** \*base)  
*De-initializes the SPI.*
- static void **SPI\_Enable** (**SPI\_Type** \*base, bool enable)  
*Enable or disable the SPI Master or Slave.*

## Status

- static **uint32\_t SPI\_GetStatusFlags** (**SPI\_Type** \*base)  
*Gets the status flag.*

## Interrupts

- static void **SPI\_EnableInterrupts** (**SPI\_Type** \*base, **uint32\_t** irqs)  
*Enables the interrupt for the SPI.*
- static void **SPI\_DisableInterrupts** (**SPI\_Type** \*base, **uint32\_t** irqs)  
*Disables the interrupt for the SPI.*

## DMA Control

- void **SPI\_EnableTxDMA** (**SPI\_Type** \*base, bool enable)  
*Enables the DMA request from SPI txFIFO.*
- void **SPI\_EnableRxDMA** (**SPI\_Type** \*base, bool enable)  
*Enables the DMA request from SPI rxFIFO.*

## Bus Operations

- **status\_t SPI\_MasterSetBaud** (**SPI\_Type** \*base, **uint32\_t** baudrate\_Bps, **uint32\_t** srcClock\_Hz)  
*Sets the baud rate for SPI transfer.*

## SPI Driver

- void **SPI\_WriteData** (SPI\_Type \*base, uint16\_t data, uint32\_t configFlags)  
*Writes a data into the SPI data register.*
- static uint32\_t **SPI\_ReadData** (SPI\_Type \*base)  
*Gets a data from the SPI data register.*
- static void **SPI\_SetTransferDelay** (SPI\_Type \*base, const spi\_delay\_config\_t \*config)  
*Set delay time for transfer.*
- void **SPI\_SetDummyData** (SPI\_Type \*base, uint8\_t dummyData)  
*Set up the dummy data.*

## Transactional

- status\_t **SPI\_MasterTransferCreateHandle** (SPI\_Type \*base, spi\_master\_handle\_t \*handle, spi\_master\_callback\_t callback, void \*userData)  
*Initializes the SPI master handle.*
- status\_t **SPI\_MasterTransferBlocking** (SPI\_Type \*base, spi\_transfer\_t \*xfer)  
*Transfers a block of data using a polling method.*
- status\_t **SPI\_MasterTransferNonBlocking** (SPI\_Type \*base, spi\_master\_handle\_t \*handle, spi\_transfer\_t \*xfer)  
*Performs a non-blocking SPI interrupt transfer.*
- status\_t **SPI\_MasterHalfDuplexTransferBlocking** (SPI\_Type \*base, spi\_half\_duplex\_transfer\_t \*xfer)  
*Transfers a block of data using a polling method.*
- status\_t **SPI\_MasterHalfDuplexTransferNonBlocking** (SPI\_Type \*base, spi\_master\_handle\_t \*handle, spi\_half\_duplex\_transfer\_t \*xfer)  
*Performs a non-blocking SPI interrupt transfer.*
- status\_t **SPI\_MasterTransferGetCount** (SPI\_Type \*base, spi\_master\_handle\_t \*handle, size\_t \*count)  
*Gets the master transfer count.*
- void **SPI\_MasterTransferAbort** (SPI\_Type \*base, spi\_master\_handle\_t \*handle)  
*SPI master aborts a transfer using an interrupt.*
- void **SPI\_MasterTransferHandleIRQ** (SPI\_Type \*base, spi\_master\_handle\_t \*handle)  
*Interrupts the handler for the SPI.*
- static status\_t **SPI\_SlaveTransferCreateHandle** (SPI\_Type \*base, spi\_slave\_handle\_t \*handle, spi\_slave\_callback\_t callback, void \*userData)  
*Initializes the SPI slave handle.*
- static status\_t **SPI\_SlaveTransferNonBlocking** (SPI\_Type \*base, spi\_slave\_handle\_t \*handle, spi\_transfer\_t \*xfer)  
*Performs a non-blocking SPI slave interrupt transfer.*
- static status\_t **SPI\_SlaveTransferGetCount** (SPI\_Type \*base, spi\_slave\_handle\_t \*handle, size\_t \*count)  
*Gets the slave transfer count.*
- static void **SPI\_SlaveTransferAbort** (SPI\_Type \*base, spi\_slave\_handle\_t \*handle)  
*SPI slave aborts a transfer using an interrupt.*
- static void **SPI\_SlaveTransferHandleIRQ** (SPI\_Type \*base, spi\_slave\_handle\_t \*handle)  
*Interrupts a handler for the SPI slave.*

## 8.3.2 Data Structure Documentation

### 8.3.2.1 struct spi\_delay\_config\_t

Note: The DLY register controls several programmable delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The maximum value of these delay time is 15.

#### Data Fields

- uint8\_t [preDelay](#)  
*Delay between SSEL assertion and the beginning of transfer.*
- uint8\_t [postDelay](#)  
*Delay between the end of transfer and SSEL deassertion.*
- uint8\_t [frameDelay](#)  
*Delay between frame to frame.*
- uint8\_t [transferDelay](#)  
*Delay between transfer to transfer.*

#### 8.3.2.1.0.10 Field Documentation

8.3.2.1.0.10.1 uint8\_t spi\_delay\_config\_t::preDelay

8.3.2.1.0.10.2 uint8\_t spi\_delay\_config\_t::postDelay

8.3.2.1.0.10.3 uint8\_t spi\_delay\_config\_t::frameDelay

8.3.2.1.0.10.4 uint8\_t spi\_delay\_config\_t::transferDelay

### 8.3.2.2 struct spi\_master\_config\_t

#### Data Fields

- bool [enableLoopback](#)  
*Enable loopback for test purpose.*
- bool [enableMaster](#)  
*Enable SPI at initialization time.*
- [spi\\_clock\\_polarity\\_t](#) polarity  
*Clock polarity.*
- [spi\\_clock\\_phase\\_t](#) phase  
*Clock phase.*
- [spi\\_shift\\_direction\\_t](#) direction  
*MSB or LSB.*
- uint32\_t [baudRate\\_Bps](#)  
*Baud Rate for SPI in Hz.*
- [spi\\_data\\_width\\_t](#) dataWidth  
*Width of the data.*
- [spi\\_ssel\\_t](#) sselNum  
*Slave select number.*
- [spi\\_spol\\_t](#) sselPol

## SPI Driver

- *Configure active CS polarity.*
- [spi\\_txfifo\\_watermark\\_t txWatermark](#)  
*txFIFO watermark*
- [spi\\_rxfifo\\_watermark\\_t rxWatermark](#)  
*rxFIFO watermark*
- [spi\\_delay\\_config\\_t delayConfig](#)  
*Delay configuration.*

### 8.3.2.2.0.11 Field Documentation

#### 8.3.2.2.0.11.1 spi\_delay\_config\_t spi\_master\_config\_t::delayConfig

### 8.3.2.3 struct spi\_slave\_config\_t

#### Data Fields

- bool [enableSlave](#)  
*Enable SPI at initialization time.*
- [spi\\_clock\\_polarity\\_t polarity](#)  
*Clock polarity.*
- [spi\\_clock\\_phase\\_t phase](#)  
*Clock phase.*
- [spi\\_shift\\_direction\\_t direction](#)  
*MSB or LSB.*
- [spi\\_data\\_width\\_t dataWidth](#)  
*Width of the data.*
- [spi\\_spol\\_t sselPol](#)  
*Configure active CS polarity.*
- [spi\\_txfifo\\_watermark\\_t txWatermark](#)  
*txFIFO watermark*
- [spi\\_rxfifo\\_watermark\\_t rxWatermark](#)  
*rxFIFO watermark*

### 8.3.2.4 struct spi\_transfer\_t

#### Data Fields

- uint8\_t \* [txData](#)  
*Send buffer.*
- uint8\_t \* [rxData](#)  
*Receive buffer.*
- uint32\_t [configFlags](#)  
*Additional option to control transfer.*
- size\_t [dataSize](#)  
*Transfer bytes.*



### 8.3.2.5 struct spi\_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.*
- bool [isPcsAssertInTransfer](#)  
*If PCS pin keep assert between transmit and receive.*
- bool [isTransmitFirst](#)  
*True for transmit first and false for receive first.*

#### 8.3.2.5.0.12 Field Documentation

##### 8.3.2.5.0.12.1 uint32\_t spi\_half\_duplex\_transfer\_t::configFlags

##### 8.3.2.5.0.12.2 bool spi\_half\_duplex\_transfer\_t::isPcsAssertInTransfer

true for assert and false for deassert.

##### 8.3.2.5.0.12.3 bool spi\_half\_duplex\_transfer\_t::isTransmitFirst

### 8.3.2.6 struct spi\_config\_t

### 8.3.2.7 struct \_spi\_master\_handle

Master handle type.

#### Data Fields

- uint8\_t \*volatile [txData](#)  
*Transfer buffer.*
- uint8\_t \*volatile [rxData](#)  
*Receive buffer.*
- volatile size\_t [txRemainingBytes](#)  
*Number of data to be transmitted [in bytes].*
- volatile size\_t [rxRemainingBytes](#)  
*Number of data to be received [in bytes].*
- volatile size\_t [toReceiveCount](#)  
*Receive data remaining in bytes.*
- size\_t [totalByteCount](#)  
*A number of transfer bytes.*
- volatile uint32\_t [state](#)

## SPI Driver

- *SPI internal state.*  
[spi\\_master\\_callback\\_t](#) callback
- *SPI callback.*  
void \* [userData](#)
- *Callback parameter.*  
uint8\_t [dataWidth](#)
- *Width of the data [Valid values: 1 to 16].*  
uint8\_t [sselNum](#)
- *Slave select number to be asserted when transferring data [Valid values: 0 to 3].*  
uint32\_t [configFlags](#)
- *Additional option to control transfer.*  
[spi\\_txfifo\\_watermark\\_t](#) txWatermark
- *txFIFO watermark*  
[spi\\_rxfifo\\_watermark\\_t](#) rxWatermark
- *rxFIFO watermark*

### 8.3.3 Macro Definition Documentation

8.3.3.1 **#define FSL\_SPI\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 2))**

8.3.3.2 **#define SPI\_DUMMYDATA (0xFFU)**

### 8.3.4 Enumeration Type Documentation

#### 8.3.4.1 enum spi\_xfer\_option\_t

Enumerator

***kSPI\_FrameDelay*** Delay chip select.

***kSPI\_FrameAssert*** When transfer ends, assert chip select.

#### 8.3.4.2 enum spi\_shift\_direction\_t

Enumerator

***kSPI\_MsbFirst*** Data transfers start with most significant bit.

***kSPI\_LsbFirst*** Data transfers start with least significant bit.

#### 8.3.4.3 enum spi\_clock\_polarity\_t

Enumerator

***kSPI\_ClockPolarityActiveHigh*** Active-high SPI clock (idles low).

***kSPI\_ClockPolarityActiveLow*** Active-low SPI clock (idles high).

#### 8.3.4.4 enum spi\_clock\_phase\_t

Enumerator

***kSPI\_ClockPhaseFirstEdge*** First edge on SCK occurs at the middle of the first cycle of a data transfer.

***kSPI\_ClockPhaseSecondEdge*** First edge on SCK occurs at the start of the first cycle of a data transfer.

#### 8.3.4.5 enum spi\_txfifo\_watermark\_t

Enumerator

***kSPI\_TxFifo0*** SPI tx watermark is empty.

***kSPI\_TxFifo1*** SPI tx watermark at 1 item.

***kSPI\_TxFifo2*** SPI tx watermark at 2 items.

***kSPI\_TxFifo3*** SPI tx watermark at 3 items.

***kSPI\_TxFifo4*** SPI tx watermark at 4 items.

***kSPI\_TxFifo5*** SPI tx watermark at 5 items.

***kSPI\_TxFifo6*** SPI tx watermark at 6 items.

***kSPI\_TxFifo7*** SPI tx watermark at 7 items.

#### 8.3.4.6 enum spi\_rxfifo\_watermark\_t

Enumerator

***kSPI\_RxFifo1*** SPI rx watermark at 1 item.

***kSPI\_RxFifo2*** SPI rx watermark at 2 items.

***kSPI\_RxFifo3*** SPI rx watermark at 3 items.

***kSPI\_RxFifo4*** SPI rx watermark at 4 items.

***kSPI\_RxFifo5*** SPI rx watermark at 5 items.

***kSPI\_RxFifo6*** SPI rx watermark at 6 items.

***kSPI\_RxFifo7*** SPI rx watermark at 7 items.

***kSPI\_RxFifo8*** SPI rx watermark at 8 items.

#### 8.3.4.7 enum spi\_data\_width\_t

Enumerator

***kSPI\_Data4Bits*** 4 bits data width

***kSPI\_Data5Bits*** 5 bits data width

***kSPI\_Data6Bits*** 6 bits data width

***kSPI\_Data7Bits*** 7 bits data width

## SPI Driver

*kSPI\_Data8Bits* 8 bits data width  
*kSPI\_Data9Bits* 9 bits data width  
*kSPI\_Data10Bits* 10 bits data width  
*kSPI\_Data11Bits* 11 bits data width  
*kSPI\_Data12Bits* 12 bits data width  
*kSPI\_Data13Bits* 13 bits data width  
*kSPI\_Data14Bits* 14 bits data width  
*kSPI\_Data15Bits* 15 bits data width  
*kSPI\_Data16Bits* 16 bits data width

### 8.3.4.8 enum spi\_ssel\_t

Enumerator

*kSPI\_Ssel0* Slave select 0.  
*kSPI\_Ssel1* Slave select 1.  
*kSPI\_Ssel2* Slave select 2.  
*kSPI\_Ssel3* Slave select 3.

### 8.3.4.9 enum \_spi\_status

Enumerator

*kStatus\_SPI\_Busy* SPI bus is busy.  
*kStatus\_SPI\_Idle* SPI is idle.  
*kStatus\_SPI\_Error* SPI error.  
*kStatus\_SPI\_BaudrateNotSupport* Baudrate is not support in current clock source.

### 8.3.4.10 enum \_spi\_interrupt\_enable

Enumerator

*kSPI\_RxLvlIrq* Rx level interrupt.  
*kSPI\_TxLvlIrq* Tx level interrupt.

### 8.3.4.11 enum \_spi\_statusflags

Enumerator

*kSPI\_TxEmptyFlag* txFifo is empty  
*kSPI\_TxNotFullFlag* txFifo is not full  
*kSPI\_RxNotEmptyFlag* rxFIFO is not empty  
*kSPI\_RxFullFlag* rxFIFO is full

## 8.3.5 Function Documentation

### 8.3.5.1 uint32\_t SPI\_GetInstance ( SPI\_Type \* *base* )

### 8.3.5.2 void SPI\_MasterGetDefaultConfig ( spi\_master\_config\_t \* *config* )

The purpose of this API is to get the configuration structure initialized for use in [SPI\\_MasterInit\(\)](#). User may use the initialized structure unchanged in [SPI\\_MasterInit\(\)](#), or modify some fields of the structure before calling [SPI\\_MasterInit\(\)](#). After calling this API, the master is ready to transfer. Example:

```
spi_master_config_t config;
SPI_MasterGetDefaultConfig(&config);
```

#### Parameters

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

### 8.3.5.3 status\_t SPI\_MasterInit ( SPI\_Type \* *base*, const spi\_master\_config\_t \* *config*, uint32\_t *srcClock\_Hz* )

The configuration structure can be filled by user from scratch, or be set with default values by [SPI\\_MasterGetDefaultConfig\(\)](#). After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
    .baudRate_Bps = 400000,
    ...
};
SPI_MasterInit(SPI0, &config);
```

#### Parameters

<i>base</i>	SPI base pointer
<i>config</i>	pointer to master configuration structure
<i>srcClock_Hz</i>	Source clock frequency.

### 8.3.5.4 void SPI\_SlaveGetDefaultConfig ( spi\_slave\_config\_t \* *config* )

The purpose of this API is to get the configuration structure initialized for use in [SPI\\_SlaveInit\(\)](#). Modify some fields of the structure before calling [SPI\\_SlaveInit\(\)](#). Example:

```
spi_slave_config_t config;
SPI_SlaveGetDefaultConfig(&config);
```

## SPI Driver

### Parameters

<i>config</i>	pointer to slave configuration structure
---------------	--

#### 8.3.5.5 `status_t SPI_SlaveInit ( SPI_Type * base, const spi_slave_config_t * config )`

The configuration structure can be filled by user from scratch or be set with default values by [SPI\\_Slave-GetDefaultConfig\(\)](#). After calling this API, the slave is ready to transfer. Example

```
spi_slave_config_t config = {  
.polarity = flexSPIClockPolarity_ActiveHigh;  
.phase = flexSPIClockPhase_FirstEdge;  
.direction = flexSPIMsbFirst;  
...  
};  
SPI_SlaveInit (SPI0, &config);
```

### Parameters

<i>base</i>	SPI base pointer
<i>config</i>	pointer to slave configuration structure

#### 8.3.5.6 `void SPI_Deinit ( SPI_Type * base )`

Calling this API resets the SPI module, gates the SPI clock. The SPI module can't work unless calling the SPI\_MasterInit/SPI\_SlaveInit to initialize module.

### Parameters

<i>base</i>	SPI base pointer
-------------	------------------

#### 8.3.5.7 `static void SPI_Enable ( SPI_Type * base, bool enable ) [inline], [static]`

### Parameters

<i>base</i>	SPI base pointer
<i>enable</i>	or disable ( true = enable, false = disable)

#### 8.3.5.8 `static uint32_t SPI_GetStatusFlags ( SPI_Type * base ) [inline], [static]`

#### Parameters

<i>base</i>	SPI base pointer
-------------	------------------

#### Returns

SPI Status, use status flag to AND [\\_spi\\_statusflags](#) could get the related status.

### 8.3.5.9 static void SPI\_EnableInterrupts ( SPI\_Type \* *base*, uint32\_t *irqs* ) [inline], [static]

#### Parameters

<i>base</i>	SPI base pointer
<i>irqs</i>	SPI interrupt source. The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• kSPI_RxLvllrq</li> <li>• kSPI_TxLvllrq</li> </ul>

### 8.3.5.10 static void SPI\_DisableInterrupts ( SPI\_Type \* *base*, uint32\_t *irqs* ) [inline], [static]

#### Parameters

<i>base</i>	SPI base pointer
<i>irqs</i>	SPI interrupt source. The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• kSPI_RxLvllrq</li> <li>• kSPI_TxLvllrq</li> </ul>

### 8.3.5.11 void SPI\_EnableTxDMA ( SPI\_Type \* *base*, bool *enable* )

#### Parameters

---

## SPI Driver

<i>base</i>	SPI base pointer
<i>enable</i>	True means enable DMA, false means disable DMA

### 8.3.5.12 void SPI\_EnableRxDMA ( SPI\_Type \* *base*, bool *enable* )

Parameters

<i>base</i>	SPI base pointer
<i>enable</i>	True means enable DMA, false means disable DMA

### 8.3.5.13 status\_t SPI\_MasterSetBaud ( SPI\_Type \* *base*, uint32\_t *baudrate\_Bps*, uint32\_t *srcClock\_Hz* )

This is only used in master.

Parameters

<i>base</i>	SPI base pointer
<i>baudrate_Bps</i>	baud rate needed in Hz.
<i>srcClock_Hz</i>	SPI source clock frequency in Hz.

### 8.3.5.14 void SPI\_WriteData ( SPI\_Type \* *base*, uint16\_t *data*, uint32\_t *configFlags* )

Parameters

<i>base</i>	SPI base pointer
<i>data</i>	needs to be write.
<i>configFlags</i>	transfer configuration options <a href="#">spi_xfer_option_t</a>

### 8.3.5.15 static uint32\_t SPI\_ReadData ( SPI\_Type \* *base* ) [inline], [static]

Parameters



<i>base</i>	SPI base pointer
-------------	------------------

## Returns

Data in the register.

### 8.3.5.16 static void SPI\_SetTransferDelay ( SPI\_Type \* *base*, const spi\_delay\_config\_t \* *config* ) [inline], [static]

the delay uint is SPI clock time, maximum value is 0xF.

## Parameters

<i>base</i>	SPI base pointer
<i>config</i>	configuration for delay option <a href="#">spi_delay_config_t</a> .

### 8.3.5.17 void SPI\_SetDummyData ( SPI\_Type \* *base*, uint8\_t *dummyData* )

## Parameters

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

### 8.3.5.18 status\_t SPI\_MasterTransferCreateHandle ( SPI\_Type \* *base*, spi\_master\_handle\_t \* *handle*, spi\_master\_callback\_t *callback*, void \* *userData* )

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

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.

## SPI Driver

<i>callback</i>	Callback function.
<i>userData</i>	User data.

### 8.3.5.19 **status\_t SPI\_MasterTransferBlocking ( SPI\_Type \* *base*, spi\_transfer\_t \* *xfer* )**

Parameters

<i>base</i>	SPI base pointer
<i>xfer</i>	pointer to spi_xfer_config_t structure

Return values

<i>kStatus_Success</i>	Successfully start a transfer.
<i>kStatus_InvalidArgument</i>	Input argument is invalid.

### 8.3.5.20 **status\_t SPI\_MasterTransferNonBlocking ( SPI\_Type \* *base*, spi\_master\_handle\_t \* *handle*, spi\_transfer\_t \* *xfer* )**

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state
<i>xfer</i>	pointer to spi_xfer_config_t structure

Return values

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

### 8.3.5.21 **status\_t SPI\_MasterHalfDuplexTransferBlocking ( SPI\_Type \* *base*, spi\_half\_duplex\_transfer\_t \* *xfer* )**

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

## Parameters

<i>base</i>	SPI base pointer
<i>xfer</i>	pointer to <a href="#">spi_half_duplex_transfer_t</a> structure

## Returns

status of status\_t.

### 8.3.5.22 status\_t SPI\_MasterHalfDuplexTransferNonBlocking ( SPI\_Type \* *base*, spi\_master\_handle\_t \* *handle*, spi\_half\_duplex\_transfer\_t \* *xfer* )

This function using polling way to do the first half transimission and using interrupts to do the second half transimission, the transfer mechanism is half-duplex. When do the second half transimission, code will return right away. When all data is transferred, the callback function is called.

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state
<i>xfer</i>	pointer to <a href="#">spi_half_duplex_transfer_t</a> structure

## Returns

status of status\_t.

### 8.3.5.23 status\_t SPI\_MasterTransferGetCount ( SPI\_Type \* *base*, spi\_master\_handle\_t \* *handle*, size\_t \* *count* )

This function gets the master transfer count.

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to the spi_master_handle_t structure which stores the transfer state.

## SPI Driver

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

Returns

status of status\_t.

**8.3.5.24 void SPI\_MasterTransferAbort ( SPI\_Type \* *base*, spi\_master\_handle\_t \* *handle* )**

This function aborts a transfer using an interrupt.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to the spi_master_handle_t structure which stores the transfer state.

**8.3.5.25 void SPI\_MasterTransferHandleIRQ ( SPI\_Type \* *base*, spi\_master\_handle\_t \* *handle* )**

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state.

**8.3.5.26 static status\_t SPI\_SlaveTransferCreateHandle ( SPI\_Type \* *base*, spi\_slave\_handle\_t \* *handle*, spi\_slave\_callback\_t *callback*, void \* *userData* ) [inline], [static]**

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

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.

<i>callback</i>	Callback function.
<i>userData</i>	User data.

**8.3.5.27 static status\_t SPI\_SlaveTransferNonBlocking ( SPI\_Type \* *base*, spi\_slave\_handle\_t \* *handle*, spi\_transfer\_t \* *xfer* ) [inline], [static]**

Note

The API returns immediately after the transfer initialization is finished.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state
<i>xfer</i>	pointer to spi_xfer_config_t structure

Return values

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

**8.3.5.28 static status\_t SPI\_SlaveTransferGetCount ( SPI\_Type \* *base*, spi\_slave\_handle\_t \* *handle*, size\_t \* *count* ) [inline], [static]**

This function gets the slave transfer count.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to the spi_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.

**8.3.5.29** `static void SPI_SlaveTransferAbort ( SPI_Type * base, spi_slave_handle_t * handle ) [inline], [static]`

This function aborts a transfer using an interrupt.

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to the spi_slave_handle_t structure which stores the transfer state.

**8.3.5.30 static void SPI\_SlaveTransferHandleIRQ ( SPI\_Type \* *base*, spi\_slave\_handle\_t \* *handle* ) [inline], [static]**

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_slave_handle_t structure which stores the transfer state

### 8.4 SPI DMA Driver

#### 8.4.1 Overview

This section describes the programming interface of the SPI DMA driver.

#### Files

- file [fsl\\_spi\\_dma.h](#)

#### Data Structures

- struct [spi\\_dma\\_handle\\_t](#)  
*SPI DMA transfer handle, users should not touch the content of the handle. [More...](#)*

#### Typedefs

- typedef void(\* [spi\\_dma\\_callback\\_t](#))(SPI\_Type \*base, spi\_dma\_handle\_t \*handle, [status\\_t](#) status, void \*userData)  
*SPI DMA callback called at the end of transfer.*

#### DMA Transactional

- [status\\_t SPI\\_MasterTransferCreateHandleDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, [spi\\_dma\\_callback\\_t](#) callback, void \*userData, [dma\\_handle\\_t](#) \*txHandle, [dma\\_handle\\_t](#) \*rxHandle)  
*Initialize the SPI master DMA handle.*
- [status\\_t SPI\\_MasterTransferDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, [spi\\_transfer\\_t](#) \*xfer)  
*Perform a non-blocking SPI transfer using DMA.*
- [status\\_t SPI\\_MasterHalfDuplexTransferDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, [spi\\_half\\_duplex\\_transfer\\_t](#) \*xfer)  
*Transfers a block of data using a DMA method.*
- static [status\\_t SPI\\_SlaveTransferCreateHandleDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, [spi\\_dma\\_callback\\_t](#) callback, void \*userData, [dma\\_handle\\_t](#) \*txHandle, [dma\\_handle\\_t](#) \*rxHandle)  
*Initialize the SPI slave DMA handle.*
- static [status\\_t SPI\\_SlaveTransferDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, [spi\\_transfer\\_t](#) \*xfer)  
*Perform a non-blocking SPI transfer using DMA.*
- void [SPI\\_MasterTransferAbortDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle)  
*Abort a SPI transfer using DMA.*
- [status\\_t SPI\\_MasterTransferGetCountDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets the master DMA transfer remaining bytes.*
- static void [SPI\\_SlaveTransferAbortDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle)



- *Abort a SPI transfer using DMA.*
- static [status\\_t SPI\\_SlaveTransferGetCountDMA](#) (SPI\_Type \*base, spi\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets the slave DMA transfer remaining bytes.*

## 8.4.2 Data Structure Documentation

### 8.4.2.1 struct \_spi\_dma\_handle

#### Data Fields

- volatile bool [txInProgress](#)  
*Send transfer finished.*
- volatile bool [rxInProgress](#)  
*Receive transfer finished.*
- [dma\\_handle\\_t](#) \* [txHandle](#)  
*DMA handler for SPI send.*
- [dma\\_handle\\_t](#) \* [rxHandle](#)  
*DMA handler for SPI receive.*
- uint8\_t [bytesPerFrame](#)  
*Bytes in a frame for SPI transfer.*
- [spi\\_dma\\_callback\\_t](#) [callback](#)  
*Callback for SPI DMA transfer.*
- void \* [userData](#)  
*User Data for SPI DMA callback.*
- uint32\_t [state](#)  
*Internal state of SPI DMA transfer.*
- size\_t [transferSize](#)  
*Bytes need to be transfer.*

## 8.4.3 Typedef Documentation

- ### 8.4.3.1 typedef void(\* spi\_dma\_callback\_t)(SPI\_Type \*base, spi\_dma\_handle\_t \*handle, status\_t status, void \*userData)

## 8.4.4 Function Documentation

- ### 8.4.4.1 status\_t SPI\_MasterTransferCreateHandleDMA ( SPI\_Type \* base, spi\_dma\_handle\_t \* handle, spi\_dma\_callback\_t callback, void \* userData, dma\_handle\_t \* txHandle, dma\_handle\_t \* rxHandle )

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

## SPI DMA Driver

### Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.
<i>callback</i>	User callback function called at the end of a transfer.
<i>userData</i>	User data for callback.
<i>txHandle</i>	DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
<i>rxHandle</i>	DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

#### 8.4.4.2 **status\_t SPI\_MasterTransferDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle*, spi\_transfer\_t \* *xfer* )**

### Note

This interface returned immediately after transfer initiates, users should call SPI\_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

### Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.
<i>xfer</i>	Pointer to dma transfer structure.

### Return values

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

#### 8.4.4.3 **status\_t SPI\_MasterHalfDuplexTransferDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle*, spi\_half\_duplex\_transfer\_t \* *xfer* )**

This function using polling way to do the first half transimission and using DMA way to do the srcond half transimission, the transfer mechanism is half-duplex. When do the second half transimission, code will return right away. When all data is transferred, the callback function is called.

## Parameters

<i>base</i>	SPI base pointer
<i>handle</i>	A pointer to the <code>spi_master_dma_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	A pointer to the <a href="#">spi_half_duplex_transfer_t</a> structure.

## Returns

status of `status_t`.

**8.4.4.4 static status\_t SPI\_SlaveTransferCreateHandleDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle*, spi\_dma\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *txHandle*, dma\_handle\_t \* *rxHandle* ) [inline], [static]**

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

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.
<i>callback</i>	User callback function called at the end of a transfer.
<i>userData</i>	User data for callback.
<i>txHandle</i>	DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
<i>rxHandle</i>	DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

**8.4.4.5 static status\_t SPI\_SlaveTransferDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle*, spi\_transfer\_t \* *xfer* ) [inline], [static]**

## Note

This interface returned immediately after transfer initiates, users should call `SPI_GetTransferStatus` to poll the transfer status to check whether SPI transfer finished.

## Parameters

## SPI DMA Driver

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.
<i>xfer</i>	Pointer to dma transfer structure.

Return values

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

### 8.4.4.6 void SPI\_MasterTransferAbortDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle* )

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.

### 8.4.4.7 status\_t SPI\_MasterTransferGetCountDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle*, size\_t \* *count* )

This function gets the master DMA transfer remaining bytes.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	A pointer to the spi_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.

### 8.4.4.8 static void SPI\_SlaveTransferAbortDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle* ) [inline], [static]

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.

**8.4.4.9 static status\_t SPI\_SlaveTransferGetCountDMA ( SPI\_Type \* *base*, spi\_dma\_handle\_t \* *handle*, size\_t \* *count* ) [inline], [static]**

This function gets the slave DMA transfer remaining bytes.

## Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	A pointer to the spi_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.

### 8.5 SPI FreeRTOS driver

#### 8.5.1 Overview

This section describes the programming interface of the SPI FreeRTOS driver.

#### Files

- file [fsl\\_spi\\_freertos.h](#)

#### Data Structures

- struct [spi\\_rtos\\_handle\\_t](#)  
*SPI FreeRTOS handle. [More...](#)*

#### SPI RTOS Operation

- [status\\_t SPI\\_RTOS\\_Init](#) ([spi\\_rtos\\_handle\\_t](#) \*handle, [SPI\\_Type](#) \*base, const [spi\\_master\\_config\\_t](#) \*masterConfig, [uint32\\_t](#) srcClock\_Hz)  
*Initializes SPI.*
- [status\\_t SPI\\_RTOS\\_Deinit](#) ([spi\\_rtos\\_handle\\_t](#) \*handle)  
*Deinitializes the SPI.*
- [status\\_t SPI\\_RTOS\\_Transfer](#) ([spi\\_rtos\\_handle\\_t](#) \*handle, [spi\\_transfer\\_t](#) \*transfer)  
*Performs SPI transfer.*

#### 8.5.2 Data Structure Documentation

##### 8.5.2.1 struct spi\_rtos\_handle\_t

#### Data Fields

- [SPI\\_Type](#) \* [base](#)  
*SPI base address.*
- [spi\\_master\\_handle\\_t](#) [drv\\_handle](#)  
*Handle of the underlying driver, treated as opaque by the RTOS layer.*
- [SemaphoreHandle\\_t](#) [mutex](#)  
*Mutex to lock the handle during a transfer.*
- [SemaphoreHandle\\_t](#) [event](#)  
*Semaphore to notify and unblock task when transfer ends.*

### 8.5.3 Function Documentation

**8.5.3.1** `status_t SPI_RTOS_Init ( spi_rtos_handle_t * handle, SPI_Type * base, const spi_master_config_t * masterConfig, uint32_t srcClock_Hz )`

This function initializes the SPI module and related RTOS context.

## SPI FreeRTOS driver

### Parameters

<i>handle</i>	The RTOS SPI handle, the pointer to an allocated space for RTOS context.
<i>base</i>	The pointer base address of the SPI instance to initialize.
<i>masterConfig</i>	Configuration structure to set-up SPI in master mode.
<i>srcClock_Hz</i>	Frequency of input clock of the SPI module.

### Returns

status of the operation.

#### 8.5.3.2 **status\_t SPI\_RTOS\_Deinit ( spi\_rtos\_handle\_t \* *handle* )**

This function deinitializes the SPI module and related RTOS context.

### Parameters

<i>handle</i>	The RTOS SPI handle.
---------------	----------------------

#### 8.5.3.3 **status\_t SPI\_RTOS\_Transfer ( spi\_rtos\_handle\_t \* *handle*, spi\_transfer\_t \* *transfer* )**

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

### Parameters

<i>handle</i>	The RTOS SPI handle.
<i>transfer</i>	Structure specifying the transfer parameters.

### Returns

status of the operation.



## Chapter 9

# USART: Universal Asynchronous Receiver/Transmitter Driver

### 9.1 Overview

The MCUXpresso SDK provides a peripheral UART driver for the Universal Synchronous Receiver/-Transmitter (USART) module of MCUXpresso SDK devices. Driver does not support synchronous mode.

The USART driver includes two parts: functional APIs and transactional APIs.

Functional APIs are used for USART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the USART peripheral and know how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. USART functional operation groups provide the functional APIs 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 `usart_handle_t` as the second parameter. Initialize the handle by calling the [USART\\_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer, which means that the functions [USART\\_TransferSendNonBlocking\(\)](#) and [USART\\_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_USART_TxIdle` and `kStatus_USART_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 [USART\\_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 [USART\\_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_USART_RxIdle`.

If the receive ring buffer is full, the upper layer is informed through a callback with the `kStatus_USART_RxRingBufferOverflow`. In the callback function, the upper layer reads data out from the ring buffer. If not, the oldest 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:

```
USART_TransferCreateHandle(USART0, &handle, USART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

## Typical use case

### 9.2 Typical use case

#### 9.2.1 USART Send/receive using a polling method

```
uint8_t ch;
USART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableTx = true;
user_config.enableRx = true;

USART_Init(USART1, &user_config, 120000000U);

while(1)
{
    USART_ReadBlocking(USART1, &ch, 1);
    USART_WriteBlocking(USART1, &ch, 1);
}
```

#### 9.2.2 USART Send/receive using an interrupt method

```
usart_handle_t g_usartHandle;
usart_config_t user_config;
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void USART_UserCallback(usart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_USART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    USART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    USART_Init(USART1, &user_config, 120000000U);
    USART_TransferCreateHandle(USART1, &g_usartHandle, USART_UserCallback, NULL);

    // Prepare to send.
    sendXfer.data = sendData;
    sendXfer.dataSize = sizeof(sendData);
    txFinished = false;

    // Send out.
    USART_TransferSendNonBlocking(USART1, &g_usartHandle, &sendXfer);
}
```

```

// Wait send finished.
while (!txFinished)
{
}

// Prepare to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData);
rxFinished = false;

// Receive.
USART_TransferReceiveNonBlocking(USART1, &g_usartHandle, &receiveXfer,
    NULL);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}

```

### 9.2.3 USART Receive using the ringbuffer feature

```

#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE 32

usart_handle_t g_usartHandle;
usart_config_t user_config;
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void USART_UserCallback(usart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_USART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    USART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    USART_Init(USART1, &user_config, 120000000U);
    USART_TransferCreateHandle(USART1, &g_usartHandle, USART_UserCallback, NULL);
    USART_TransferStartRingBuffer(USART1, &g_usartHandle, ringBuffer,
        RING_BUFFER_SIZE);
    // Now the RX is working in background, receive in to ring buffer.

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = sizeof(receiveData);
}

```

## Typical use case

```
rxFinished = false;

// Receive.
USART_TransferReceiveNonBlocking(USART1, &g_usartHandle, &receiveXfer);

if (bytesRead == RX_DATA_SIZE) /* Have read enough data. */
{
    ;
}
else
{
    if (bytesRead) /* Received some data, process first. */
    {
        ;
    }

    // Wait receive finished.
    while (!rxFinished)
    {
    }
}

// ...
}
```

### 9.2.4 USART Send/Receive using the DMA method

```
usart_handle_t g_usartHandle;
dma_handle_t g_usartTxDmaHandle;
dma_handle_t g_usartRxDmaHandle;
usart_config_t user_config;
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void USART_UserCallback(usart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_USART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    USART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    USART_Init(USART1, &user_config, 120000000U);

    // Set up the DMA
```

```

DMA_Init(DMA0);
DMA_EnableChannel(DMA0, USART_TX_DMA_CHANNEL);
DMA_EnableChannel(DMA0, USART_RX_DMA_CHANNEL);

DMA_CreateHandle(&g_usartTxDmaHandle, DMA0, USART_TX_DMA_CHANNEL);
DMA_CreateHandle(&g_usartRxDmaHandle, DMA0, USART_RX_DMA_CHANNEL);

USART_TransferCreateHandleDMA(USART1, &g_usartHandle, USART_UserCallback,
    NULL, &g_usartTxDmaHandle, &g_usartRxDmaHandle);

// Prepare to send.
sendXfer.data = sendData;
sendXfer.dataSize = sizeof(sendData);
txFinished = false;

// Send out.
USART_TransferSendDMA(USART1, &g_usartHandle, &sendXfer);

// Wait send finished.
while (!txFinished)
{
}

// Prepare to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData);
rxFinished = false;

// Receive.
USART_TransferReceiveDMA(USART1, &g_usartHandle, &receiveXfer);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}

```

## Modules

- [USART DMA Driver](#)
- [USART Driver](#)
- [USART FreeRTOS Driver](#)

### 9.3 USART Driver

#### 9.3.1 Overview

##### Data Structures

- struct `usart_config_t`  
*USART configuration structure. [More...](#)*
- struct `usart_transfer_t`  
*USART transfer structure. [More...](#)*
- struct `usart_handle_t`  
*USART handle structure. [More...](#)*

##### Typedefs

- typedef `void(* usart_transfer_callback_t)(USART_Type *base, usart_handle_t *handle, status_t status, void *userData)`  
*USART transfer callback function.*

##### Enumerations

- enum `_usart_status` {  
    `kStatus_USART_TxBusy` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 0),  
    `kStatus_USART_RxBusy` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 1),  
    `kStatus_USART_TxIdle` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 2),  
    `kStatus_USART_RxIdle` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 3),  
    `kStatus_USART_TxError` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 7),  
    `kStatus_USART_RxError` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 9),  
    `kStatus_USART_RxRingBufferOverrun` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 8),  
    `kStatus_USART_NoiseError` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 10),  
    `kStatus_USART_FramingError` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 11),  
    `kStatus_USART_ParityError` = MAKE\_STATUS(kStatusGroup\_LPC\_USART, 12),  
    `kStatus_USART_BaudrateNotSupport` }  
    *Error codes for the USART driver.*
- enum `usart_parity_mode_t` {  
    `kUSART_ParityDisabled` = 0x0U,  
    `kUSART_ParityEven` = 0x2U,  
    `kUSART_ParityOdd` = 0x3U }  
    *USART parity mode.*
- enum `usart_stop_bit_count_t` {  
    `kUSART_OneStopBit` = 0U,  
    `kUSART_TwoStopBit` = 1U }  
    *USART stop bit count.*
- enum `usart_data_len_t` {  
    `kUSART_7BitsPerChar` = 0U,

- `kUSART_8BitsPerChar = 1U` }  
*USART data size.*
- enum `usart_txfifo_watermark_t` {  
`kUSART_TxFifo0 = 0,`  
`kUSART_TxFifo1 = 1,`  
`kUSART_TxFifo2 = 2,`  
`kUSART_TxFifo3 = 3,`  
`kUSART_TxFifo4 = 4,`  
`kUSART_TxFifo5 = 5,`  
`kUSART_TxFifo6 = 6,`  
`kUSART_TxFifo7 = 7` }  
*txFIFO watermark values*
- enum `usart_rxfifo_watermark_t` {  
`kUSART_RxFifo1 = 0,`  
`kUSART_RxFifo2 = 1,`  
`kUSART_RxFifo3 = 2,`  
`kUSART_RxFifo4 = 3,`  
`kUSART_RxFifo5 = 4,`  
`kUSART_RxFifo6 = 5,`  
`kUSART_RxFifo7 = 6,`  
`kUSART_RxFifo8 = 7` }  
*rxFIFO watermark values*
- enum `_usart_interrupt_enable`  
*USART interrupt configuration structure, default settings all disabled.*
- enum `_usart_flags` {  
`kUSART_TxError = (USART_FIFOSTAT_TXERR_MASK),`  
`kUSART_RxError = (USART_FIFOSTAT_RXERR_MASK),`  
`kUSART_TxFifoEmptyFlag = (USART_FIFOSTAT_TXEMPTY_MASK),`  
`kUSART_TxFifoNotFullFlag = (USART_FIFOSTAT_TXNOTFULL_MASK),`  
`kUSART_RxFifoNotEmptyFlag = (USART_FIFOSTAT_RXNOTEMPTY_MASK),`  
`kUSART_RxFifoFullFlag = (USART_FIFOSTAT_RXFULL_MASK)` }  
*USART status flags.*

## Functions

- `uint32_t USART_GetInstance (USART_Type *base)`  
*Returns instance number for USART peripheral base address.*

## Driver version

- `#define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`  
*USART driver version 2.0.0.*

### Initialization and deinitialization

- **status\_t USART\_Init** (USART\_Type \*base, const **usart\_config\_t** \*config, uint32\_t srcClock\_Hz)  
*Initializes a USART instance with user configuration structure and peripheral clock.*
- **void USART\_Deinit** (USART\_Type \*base)  
*Deinitializes a USART instance.*
- **void USART\_GetDefaultConfig** (**usart\_config\_t** \*config)  
*Gets the default configuration structure.*
- **status\_t USART\_SetBaudRate** (USART\_Type \*base, uint32\_t baudrate\_Bps, uint32\_t srcClock\_Hz)  
*Sets the USART instance baud rate.*

### Status

- **static uint32\_t USART\_GetStatusFlags** (USART\_Type \*base)  
*Get USART status flags.*
- **static void USART\_ClearStatusFlags** (USART\_Type \*base, uint32\_t mask)  
*Clear USART status flags.*

### Interrupts

- **static void USART\_EnableInterrupts** (USART\_Type \*base, uint32\_t mask)  
*Enables USART interrupts according to the provided mask.*
- **static void USART\_DisableInterrupts** (USART\_Type \*base, uint32\_t mask)  
*Disables USART interrupts according to a provided mask.*
- **static uint32\_t USART\_GetEnabledInterrupts** (USART\_Type \*base)  
*Returns enabled USART interrupts.*
- **static void USART\_EnableTxDMA** (USART\_Type \*base, bool enable)  
*Enable DMA for Tx.*
- **static void USART\_EnableRxDMA** (USART\_Type \*base, bool enable)  
*Enable DMA for Rx.*

### Bus Operations

- **static void USART\_WriteByte** (USART\_Type \*base, uint8\_t data)  
*Writes to the FIFOWR register.*
- **static uint8\_t USART\_ReadByte** (USART\_Type \*base)  
*Reads the FIFORD register directly.*
- **void USART\_WriteBlocking** (USART\_Type \*base, const uint8\_t \*data, size\_t length)  
*Writes to the TX register using a blocking method.*
- **status\_t USART\_ReadBlocking** (USART\_Type \*base, uint8\_t \*data, size\_t length)  
*Read RX data register using a blocking method.*



## Transactional

- [status\\_t USART\\_TransferCreateHandle](#) (USART\_Type \*base, usart\_handle\_t \*handle, [usart\\_transfer\\_callback\\_t](#) callback, void \*userData)  
*Initializes the USART handle.*
- [status\\_t USART\\_TransferSendNonBlocking](#) (USART\_Type \*base, usart\_handle\_t \*handle, [usart\\_transfer\\_t](#) \*xfer)  
*Transmits a buffer of data using the interrupt method.*
- void [USART\\_TransferStartRingBuffer](#) (USART\_Type \*base, usart\_handle\_t \*handle, uint8\_t \*ringBuffer, size\_t ringBufferSize)  
*Sets up the RX ring buffer.*
- void [USART\\_TransferStopRingBuffer](#) (USART\_Type \*base, usart\_handle\_t \*handle)  
*Aborts the background transfer and uninstalls the ring buffer.*
- size\_t [USART\\_TransferGetRxRingBufferLength](#) (usart\_handle\_t \*handle)  
*Get the length of received data in RX ring buffer.*
- void [USART\\_TransferAbortSend](#) (USART\_Type \*base, usart\_handle\_t \*handle)  
*Aborts the interrupt-driven data transmit.*
- [status\\_t USART\\_TransferGetSendCount](#) (USART\_Type \*base, usart\_handle\_t \*handle, uint32\_t \*count)  
*Get the number of bytes that have been written to USART TX register.*
- [status\\_t USART\\_TransferReceiveNonBlocking](#) (USART\_Type \*base, usart\_handle\_t \*handle, [usart\\_transfer\\_t](#) \*xfer, size\_t \*receivedBytes)  
*Receives a buffer of data using an interrupt method.*
- void [USART\\_TransferAbortReceive](#) (USART\_Type \*base, usart\_handle\_t \*handle)  
*Aborts the interrupt-driven data receiving.*
- [status\\_t USART\\_TransferGetReceiveCount](#) (USART\_Type \*base, usart\_handle\_t \*handle, uint32\_t \*count)  
*Get the number of bytes that have been received.*
- void [USART\\_TransferHandleIRQ](#) (USART\_Type \*base, usart\_handle\_t \*handle)  
*USART IRQ handle function.*

## 9.3.2 Data Structure Documentation

### 9.3.2.1 struct usart\_config\_t

#### Data Fields

- uint32\_t [baudRate\\_Bps](#)  
*USART baud rate.*
- [usart\\_parity\\_mode\\_t](#) parityMode  
*Parity mode, disabled (default), even, odd.*
- [usart\\_stop\\_bit\\_count\\_t](#) stopBitCount  
*Number of stop bits, 1 stop bit (default) or 2 stop bits.*
- [usart\\_data\\_len\\_t](#) bitCountPerChar  
*Data length - 7 bit, 8 bit.*
- bool [loopback](#)  
*Enable peripheral loopback.*
- bool [enableRx](#)

## USART Driver

- *Enable RX.*  
• bool [enableTx](#)  
*Enable TX.*
- [usart\\_txfifo\\_watermark\\_t txWatermark](#)  
*txFIFO watermark*
- [usart\\_rxfifo\\_watermark\\_t rxWatermark](#)  
*rxFIFO watermark*

### 9.3.2.2 struct usart\_transfer\_t

#### Data Fields

- uint8\_t \* [data](#)  
*The buffer of data to be transfer.*
- size\_t [dataSize](#)  
*The byte count to be transfer.*

#### 9.3.2.2.0.13 Field Documentation

##### 9.3.2.2.0.13.1 uint8\_t\* usart\_transfer\_t::data

##### 9.3.2.2.0.13.2 size\_t usart\_transfer\_t::dataSize

### 9.3.2.3 struct \_usart\_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.*
- [usart\\_transfer\\_callback\\_t callback](#)  
*Callback function.*
- void \* [userData](#)

- *USART callback function parameter.*  
volatile uint8\_t txState  
*TX transfer state.*
- volatile uint8\_t rxState  
*RX transfer state.*
- usart\_txfifo\_watermark\_t txWatermark  
*txFIFO watermark*
- usart\_rxfifo\_watermark\_t rxWatermark  
*rxFIFO watermark*

## USART Driver

### 9.3.2.3.0.14 Field Documentation

9.3.2.3.0.14.1 `uint8_t* volatile usart_handle_t::txData`

9.3.2.3.0.14.2 `volatile size_t usart_handle_t::txDataSize`

9.3.2.3.0.14.3 `size_t usart_handle_t::txDataSizeAll`

9.3.2.3.0.14.4 `uint8_t* volatile usart_handle_t::rxData`

9.3.2.3.0.14.5 `volatile size_t usart_handle_t::rxDataSize`

9.3.2.3.0.14.6 `size_t usart_handle_t::rxDataSizeAll`

9.3.2.3.0.14.7 `uint8_t* usart_handle_t::rxRingBuffer`

9.3.2.3.0.14.8 `size_t usart_handle_t::rxRingBufferSize`

9.3.2.3.0.14.9 `volatile uint16_t usart_handle_t::rxRingBufferHead`

9.3.2.3.0.14.10 `volatile uint16_t usart_handle_t::rxRingBufferTail`

9.3.2.3.0.14.11 `usart_transfer_callback_t usart_handle_t::callback`

9.3.2.3.0.14.12 `void* usart_handle_t::userData`

9.3.2.3.0.14.13 `volatile uint8_t usart_handle_t::txState`

### 9.3.3 Macro Definition Documentation

9.3.3.1 `#define FSL_USART_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

### 9.3.4 Typedef Documentation

9.3.4.1 `typedef void(* usart_transfer_callback_t)(USART_Type *base, usart_handle_t *handle, status_t status, void *userData)`

### 9.3.5 Enumeration Type Documentation

#### 9.3.5.1 `enum _usart_status`

Enumerator

*kStatus\_USART\_TxBusy* Transmitter is busy.

*kStatus\_USART\_RxBusy* Receiver is busy.

*kStatus\_USART\_TxIdle* USART transmitter is idle.

*kStatus\_USART\_RxIdle* USART receiver is idle.

*kStatus\_USART\_TxError* Error happens on txFIFO.

***kStatus\_USART\_RxError*** Error happens on rxFIFO.  
***kStatus\_USART\_RxRingBufferOverrun*** Error happens on rx ring buffer.  
***kStatus\_USART\_NoiseError*** USART noise error.  
***kStatus\_USART\_FramingError*** USART framing error.  
***kStatus\_USART\_ParityError*** USART parity error.  
***kStatus\_USART\_BaudrateNotSupport*** Baudrate is not support in current clock source.

### 9.3.5.2 enum usart\_parity\_mode\_t

Enumerator

***kUSART\_ParityDisabled*** Parity disabled.  
***kUSART\_ParityEven*** Parity enabled, type even, bit setting: PE|PT = 10.  
***kUSART\_ParityOdd*** Parity enabled, type odd, bit setting: PE|PT = 11.

### 9.3.5.3 enum usart\_stop\_bit\_count\_t

Enumerator

***kUSART\_OneStopBit*** One stop bit.  
***kUSART\_TwoStopBit*** Two stop bits.

### 9.3.5.4 enum usart\_data\_len\_t

Enumerator

***kUSART\_7BitsPerChar*** Seven bit mode.  
***kUSART\_8BitsPerChar*** Eight bit mode.

### 9.3.5.5 enum usart\_txfifo\_watermark\_t

Enumerator

***kUSART\_TxFifo0*** USART tx watermark is empty.  
***kUSART\_TxFifo1*** USART tx watermark at 1 item.  
***kUSART\_TxFifo2*** USART tx watermark at 2 items.  
***kUSART\_TxFifo3*** USART tx watermark at 3 items.  
***kUSART\_TxFifo4*** USART tx watermark at 4 items.  
***kUSART\_TxFifo5*** USART tx watermark at 5 items.  
***kUSART\_TxFifo6*** USART tx watermark at 6 items.  
***kUSART\_TxFifo7*** USART tx watermark at 7 items.

### 9.3.5.6 enum usart\_rxfifo\_watermark\_t

Enumerator

***kUSART\_RxFifo1*** USART rx watermark at 1 item.  
***kUSART\_RxFifo2*** USART rx watermark at 2 items.  
***kUSART\_RxFifo3*** USART rx watermark at 3 items.  
***kUSART\_RxFifo4*** USART rx watermark at 4 items.  
***kUSART\_RxFifo5*** USART rx watermark at 5 items.  
***kUSART\_RxFifo6*** USART rx watermark at 6 items.  
***kUSART\_RxFifo7*** USART rx watermark at 7 items.  
***kUSART\_RxFifo8*** USART rx watermark at 8 items.

### 9.3.5.7 enum \_usart\_flags

This provides constants for the USART status flags for use in the USART functions.

Enumerator

***kUSART\_TxError*** TEERR bit, sets if TX buffer is error.  
***kUSART\_RxError*** RXERR bit, sets if RX buffer is error.  
***kUSART\_TxFifoEmptyFlag*** TXEMPTY bit, sets if TX buffer is empty.  
***kUSART\_TxFifoNotFullFlag*** TXNOTFULL bit, sets if TX buffer is not full.  
***kUSART\_RxFifoNotEmptyFlag*** RXNOEMPTY bit, sets if RX buffer is not empty.  
***kUSART\_RxFifoFullFlag*** RXFULL bit, sets if RX buffer is full.

## 9.3.6 Function Documentation

### 9.3.6.1 uint32\_t USART\_GetInstance ( USART\_Type \* *base* )

### 9.3.6.2 status\_t USART\_Init ( USART\_Type \* *base*, const usart\_config\_t \* *config*, uint32\_t *srcClock\_Hz* )

This function configures the USART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the [USART\\_GetDefaultConfig\(\)](#) function. Example below shows how to use this API to configure USART.

```
* usart_config_t usartConfig;  
* usartConfig.baudRate_Bps = 115200U;  
* usartConfig.parityMode = kUSART_ParityDisabled;  
* usartConfig.stopBitCount = kUSART_OneStopBit;  
* USART_Init(USART1, &usartConfig, 20000000U);  
*
```

## Parameters

<i>base</i>	USART peripheral base address.
<i>config</i>	Pointer to user-defined configuration structure.
<i>srcClock_Hz</i>	USART clock source frequency in HZ.

## Return values

<i>kStatus_USART_BaudrateNotSupport</i>	Baudrate is not support in current clock source.
<i>kStatus_InvalidArgument</i>	USART base address is not valid
<i>kStatus_Success</i>	Status USART initialize succeed

**9.3.6.3 void USART\_Deinit ( USART\_Type \* *base* )**

This function waits for TX complete, disables TX and RX, and disables the USART clock.

## Parameters

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

**9.3.6.4 void USART\_GetDefaultConfig ( usart\_config\_t \* *config* )**

This function initializes the USART configuration structure to a default value. The default values are: usartConfig->baudRate\_Bps = 115200U; usartConfig->parityMode = kUSART\_ParityDisabled; usartConfig->stopBitCount = kUSART\_OneStopBit; usartConfig->bitCountPerChar = kUSART\_8BitsPerChar; usartConfig->loopback = false; usartConfig->enableTx = false; usartConfig->enableRx = false;

## Parameters

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

**9.3.6.5 status\_t USART\_SetBaudRate ( USART\_Type \* *base*, uint32\_t *baudrate\_Bps*, uint32\_t *srcClock\_Hz* )**

This function configures the USART module baud rate. This function is used to update the USART module baud rate after the USART module is initialized by the USART\_Init.

```
* USART_SetBaudRate(USART1, 115200U, 200000000U);
*
```

## USART Driver

### Parameters

<i>base</i>	USART peripheral base address.
<i>baudrate_Bps</i>	USART baudrate to be set.
<i>srcClock_Hz</i>	USART clock source frequency in HZ.

### Return values

<i>kStatus_USART_-BaudrateNotSupport</i>	Baudrate is not support in current clock source.
<i>kStatus_Success</i>	Set baudrate succeed.
<i>kStatus_InvalidArgument</i>	One or more arguments are invalid.

#### 9.3.6.6 static uint32\_t USART\_GetStatusFlags ( USART\_Type \* *base* ) [inline], [static]

This function get all USART status flags, the flags are returned as the logical OR value of the enumerators [\\_usart\\_flags](#). To check a specific status, compare the return value with enumerators in [\\_usart\\_flags](#). For example, to check whether the TX is empty:

```
*  if (kUSART_TxFifoNotFullFlag &  
    USART_GetStatusFlags(USART1))  
*  {  
*      ...  
*  }  
*
```

### Parameters

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

### Returns

USART status flags which are ORed by the enumerators in the [\\_usart\\_flags](#).

#### 9.3.6.7 static void USART\_ClearStatusFlags ( USART\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function clear supported USART status flags. Flags that can be cleared or set are: kUSART\_TxError, kUSART\_RxError. For example:

```
*  USART_ClearStatusFlags(USART1, kUSART_TxError |  
    kUSART_RxError)  
*
```



## Parameters

<i>base</i>	USART peripheral base address.
<i>mask</i>	status flags to be cleared.

### 9.3.6.8 static void USART\_EnableInterrupts ( USART\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function enables the USART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [\\_usart\\_interrupt\\_enable](#). For example, to enable TX empty interrupt and RX full interrupt:

```
*  USART_EnableInterrupts(USART1, kUSART_TxLevelInterruptEnable |
*  kUSART_RxLevelInterruptEnable);
```

## Parameters

<i>base</i>	USART peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of <a href="#">_usart_interrupt_enable</a> .

### 9.3.6.9 static void USART\_DisableInterrupts ( USART\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function disables the USART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See [\\_usart\\_interrupt\\_enable](#). This example shows how to disable the TX empty interrupt and RX full interrupt:

```
*  USART_DisableInterrupts(USART1, kUSART_TxLevelInterruptEnable |
*  kUSART_RxLevelInterruptEnable);
```

## Parameters

<i>base</i>	USART peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of <a href="#">_usart_interrupt_enable</a> .

### 9.3.6.10 static uint32\_t USART\_GetEnabledInterrupts ( USART\_Type \* *base* ) [inline], [static]

This function returns the enabled USART interrupts.

## USART Driver

### Parameters

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

### 9.3.6.11 **static void USART\_WriteByte ( USART\_Type \* *base*, uint8\_t *data* ) [inline], [static]**

This function writes data to the txFIFO directly. The upper layer must ensure that txFIFO has space for data to write before calling this function.

### Parameters

<i>base</i>	USART peripheral base address.
<i>data</i>	The byte to write.

### 9.3.6.12 **static uint8\_t USART\_ReadByte ( USART\_Type \* *base* ) [inline], [static]**

This function reads data from the rxFIFO directly. The upper layer must ensure that the rxFIFO is not empty before calling this function.

### Parameters

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

### Returns

The byte read from USART data register.

### 9.3.6.13 **void USART\_WriteBlocking ( USART\_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.

### Parameters

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

<i>data</i>	Start address of the data to write.
<i>length</i>	Size of the data to write.

#### 9.3.6.14 **status\_t USART\_ReadBlocking ( USART\_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 read data from the TX register.

Parameters

<i>base</i>	USART peripheral base address.
<i>data</i>	Start address of the buffer to store the received data.
<i>length</i>	Size of the buffer.

Return values

<i>kStatus_USART_FramingError</i>	Receiver overrun happened while receiving data.
<i>kStatus_USART_ParityError</i>	Noise error happened while receiving data.
<i>kStatus_USART_NoiseError</i>	Framing error happened while receiving data.
<i>kStatus_USART_RxError</i>	Overflow or underflow rxFIFO happened.
<i>kStatus_Success</i>	Successfully received all data.

#### 9.3.6.15 **status\_t USART\_TransferCreateHandle ( USART\_Type \* *base*, usart\_handle\_t \* *handle*, usart\_transfer\_callback\_t *callback*, void \* *userData* )**

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

Parameters

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

## USART Driver

<i>handle</i>	USART handle pointer.
<i>callback</i>	The callback function.
<i>userData</i>	The parameter of the callback function.

### 9.3.6.16 **status\_t USART\_TransferSendNonBlocking ( USART\_Type \* *base*, usart\_handle\_t \* *handle*, usart\_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 IRQ handler, the USART driver calls the callback function and passes the [kStatus\\_USART\\_TxIdle](#) as status parameter.

#### Note

The [kStatus\\_USART\\_TxIdle](#) is passed to the upper layer when all data is written to the TX register. However it does not ensure that all data are sent out. Before disabling the TX, check the [kUSART\\_TransmissionCompleteFlag](#) to ensure that the TX is finished.

#### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.
<i>xfer</i>	USART transfer structure. See <a href="#">usart_transfer_t</a> .

#### Return values

<i>kStatus_Success</i>	Successfully start the data transmission.
<i>kStatus_USART_TxBusy</i>	Previous transmission still not finished, data not all written to TX register yet.
<i>kStatus_InvalidArgument</i>	Invalid argument.

### 9.3.6.17 **void USART\_TransferStartRingBuffer ( USART\_Type \* *base*, usart\_handle\_t \* *handle*, uint8\_t \* *ringBuffer*, size\_t *ringBufferSize* )**

This function sets up the RX ring buffer to a specific USART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the [USART\\_TransferReceiveNonBlocking\(\)](#) API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

### Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART 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.

#### 9.3.6.18 void USART\_TransferStopRingBuffer ( USART\_Type \* *base*, usart\_handle\_t \* *handle* )

This function aborts the background transfer and uninstalls the ring buffer.

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.

#### 9.3.6.19 size\_t USART\_TransferGetRxRingBufferLength ( usart\_handle\_t \* *handle* )

### Parameters

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

### Returns

Length of received data in RX ring buffer.

#### 9.3.6.20 void USART\_TransferAbortSend ( USART\_Type \* *base*, usart\_handle\_t \* *handle* )

This function aborts the interrupt driven data sending. The user can get the `remainBtyes` to find out how many bytes are still not sent out.

## USART Driver

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.

#### 9.3.6.21 **status\_t USART\_TransferGetSendCount ( USART\_Type \* *base*, usart\_handle\_t \* *handle*, uint32\_t \* *count* )**

This function gets the number of bytes that have been written to USART TX register by interrupt method.

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.
<i>count</i>	Send bytes count.

### Return values

<i>kStatus_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

#### 9.3.6.22 **status\_t USART\_TransferReceiveNonBlocking ( USART\_Type \* *base*, usart\_handle\_t \* *handle*, usart\_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 USART driver. When the new data arrives, the receive request is serviced first. When all data is received, the USART driver notifies the upper layer through a callback function and passes the status parameter [kStatus\\_USART\\_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 USART 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>	USART peripheral base address.
<i>handle</i>	USART handle pointer.
<i>xfer</i>	USART transfer structure, see <a href="#">usart_transfer_t</a> .
<i>receivedBytes</i>	Bytes received from the ring buffer directly.

#### Return values

<i>kStatus_Success</i>	Successfully queue the transfer into transmit queue.
<i>kStatus_USART_RxBusy</i>	Previous receive request is not finished.
<i>kStatus_InvalidArgument</i>	Invalid argument.

#### 9.3.6.23 void USART\_TransferAbortReceive ( USART\_Type \* *base*, usart\_handle\_t \* *handle* )

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to find out how many bytes not received yet.

#### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.

#### 9.3.6.24 status\_t USART\_TransferGetReceiveCount ( USART\_Type \* *base*, usart\_handle\_t \* *handle*, uint32\_t \* *count* )

This function gets the number of bytes that have been received.

#### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.
<i>count</i>	Receive bytes count.

## USART Driver

Return values

<i>kStatus_NoTransferInProgress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter count;

### 9.3.6.25 void USART\_TransferHandleIRQ ( USART\_Type \* *base*, usart\_handle\_t \* *handle* )

This function handles the USART transmit and receive IRQ request.

Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.



## 9.4 USART DMA Driver

### 9.4.1 Overview

#### Files

- file [fsl\\_usart\\_dma.h](#)

#### Data Structures

- struct [usart\\_dma\\_handle\\_t](#)  
*UART DMA handle. [More...](#)*

#### Typedefs

- typedef void(\* [usart\\_dma\\_transfer\\_callback\\_t](#))(USART\_Type \*base, usart\_dma\_handle\_t \*handle, [status\\_t](#) status, void \*userData)  
*UART transfer callback function.*

#### DMA transactional

- [status\\_t USART\\_TransferCreateHandleDMA](#) (USART\_Type \*base, usart\_dma\_handle\_t \*handle, [usart\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, [dma\\_handle\\_t](#) \*txDmaHandle, [dma\\_handle\\_t](#) \*rxDmaHandle)  
*Initializes the USART handle which is used in transactional functions.*
- [status\\_t USART\\_TransferSendDMA](#) (USART\_Type \*base, usart\_dma\_handle\_t \*handle, [usart\\_transfer\\_t](#) \*xfer)  
*Sends data using DMA.*
- [status\\_t USART\\_TransferReceiveDMA](#) (USART\_Type \*base, usart\_dma\_handle\_t \*handle, [usart\\_transfer\\_t](#) \*xfer)  
*Receives data using DMA.*
- void [USART\\_TransferAbortSendDMA](#) (USART\_Type \*base, usart\_dma\_handle\_t \*handle)  
*Aborts the sent data using DMA.*
- void [USART\\_TransferAbortReceiveDMA](#) (USART\_Type \*base, usart\_dma\_handle\_t \*handle)  
*Aborts the received data using DMA.*
- [status\\_t USART\\_TransferGetReceiveCountDMA](#) (USART\_Type \*base, usart\_dma\_handle\_t \*handle, uint32\_t \*count)  
*Get the number of bytes that have been received.*

### 9.4.2 Data Structure Documentation

#### 9.4.2.1 struct \_usart\_dma\_handle

##### Data Fields

- USART\_Type \* [base](#)  
*USART peripheral base address.*
- [usart\\_dma\\_transfer\\_callback\\_t](#) [callback](#)  
*Callback function.*
- void \* [userData](#)  
*USART 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.*

#### 9.4.2.1.0.15 Field Documentation

9.4.2.1.0.15.1 `USART_Type* usart_dma_handle_t::base`

9.4.2.1.0.15.2 `usart_dma_transfer_callback_t usart_dma_handle_t::callback`

9.4.2.1.0.15.3 `void* usart_dma_handle_t::userData`

9.4.2.1.0.15.4 `size_t usart_dma_handle_t::rxDataSizeAll`

9.4.2.1.0.15.5 `size_t usart_dma_handle_t::txDataSizeAll`

9.4.2.1.0.15.6 `dma_handle_t* usart_dma_handle_t::txDmaHandle`

9.4.2.1.0.15.7 `dma_handle_t* usart_dma_handle_t::rxDmaHandle`

9.4.2.1.0.15.8 `volatile uint8_t usart_dma_handle_t::txState`

#### 9.4.3 Typedef Documentation

9.4.3.1 `typedef void(* usart_dma_transfer_callback_t)(USART_Type *base,  
usart_dma_handle_t *handle, status_t status, void *userData)`

#### 9.4.4 Function Documentation

9.4.4.1 `status_t USART_TransferCreateHandleDMA ( USART_Type * base,  
usart_dma_handle_t * handle, usart_dma_transfer_callback_t callback, void *  
userData, dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle )`

## USART DMA Driver

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	Pointer to usart_dma_handle_t structure.
<i>callback</i>	Callback function.
<i>userData</i>	User data.
<i>txDmaHandle</i>	User-requested DMA handle for TX DMA transfer.
<i>rxDmaHandle</i>	User-requested DMA handle for RX DMA transfer.

#### 9.4.4.2 **status\_t USART\_TransferSendDMA ( USART\_Type \* *base*, usart\_dma\_handle\_t \* *handle*, usart\_transfer\_t \* *xfer* )**

This function sends data using DMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.
<i>xfer</i>	USART DMA transfer structure. See <a href="#">usart_transfer_t</a> .

### Return values

<i>kStatus_Success</i>	if succeed, others failed.
<i>kStatus_USART_TxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

#### 9.4.4.3 **status\_t USART\_TransferReceiveDMA ( USART\_Type \* *base*, usart\_dma\_handle\_t \* *handle*, usart\_transfer\_t \* *xfer* )**

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	Pointer to usart_dma_handle_t structure.
<i>xfer</i>	USART DMA transfer structure. See <a href="#">usart_transfer_t</a> .

Return values

<i>kStatus_Success</i>	if succeed, others failed.
<i>kStatus_USART_RxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

#### 9.4.4.4 void USART\_TransferAbortSendDMA ( USART\_Type \* *base*, usart\_dma\_handle\_t \* *handle* )

This function aborts send data using DMA.

Parameters

<i>base</i>	USART peripheral base address
<i>handle</i>	Pointer to usart_dma_handle_t structure

#### 9.4.4.5 void USART\_TransferAbortReceiveDMA ( USART\_Type \* *base*, usart\_dma\_handle\_t \* *handle* )

This function aborts the received data using DMA.

Parameters

<i>base</i>	USART peripheral base address
<i>handle</i>	Pointer to usart_dma_handle_t structure

#### 9.4.4.6 status\_t USART\_TransferGetReceiveCountDMA ( USART\_Type \* *base*, usart\_dma\_handle\_t \* *handle*, uint32\_t \* *count* )

This function gets the number of bytes that have been received.

## USART DMA Driver

### Parameters

<i>base</i>	USART peripheral base address.
<i>handle</i>	USART handle pointer.
<i>count</i>	Receive bytes count.

### Return values

<i>kStatus_NoTransferInProgress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

## 9.5 USART FreeRTOS Driver

### 9.5.1 Overview

#### Files

- file [fsl\\_usart\\_freertos.h](#)

#### Data Structures

- struct [rtos\\_usart\\_config](#)  
*FLEX USART configuration structure. [More...](#)*
- struct [usart\\_rtos\\_handle\\_t](#)  
*FLEX USART FreeRTOS handle. [More...](#)*

#### USART RTOS Operation

- int [USART\\_RTOS\\_Init](#) ([usart\\_rtos\\_handle\\_t](#) \*handle, [usart\\_handle\\_t](#) \*t\_handle, const struct [rtos\\_usart\\_config](#) \*cfg)  
*Initializes a USART instance for operation in RTOS.*
- int [USART\\_RTOS\\_Deinit](#) ([usart\\_rtos\\_handle\\_t](#) \*handle)  
*Deinitializes a USART instance for operation.*

#### USART transactional Operation

- int [USART\\_RTOS\\_Send](#) ([usart\\_rtos\\_handle\\_t](#) \*handle, const [uint8\\_t](#) \*buffer, [uint32\\_t](#) length)  
*Sends data in the background.*
- int [USART\\_RTOS\\_Receive](#) ([usart\\_rtos\\_handle\\_t](#) \*handle, [uint8\\_t](#) \*buffer, [uint32\\_t](#) length, [size\\_t](#) \*received)  
*Receives data.*

### 9.5.2 Data Structure Documentation

#### 9.5.2.1 struct [rtos\\_usart\\_config](#)

##### Data Fields

- [USART\\_Type](#) \* [base](#)  
*USART base address.*
- [uint32\\_t](#) [srcclk](#)  
*USART source clock in Hz.*
- [uint32\\_t](#) [baudrate](#)  
*Desired communication speed.*
- [usart\\_parity\\_mode\\_t](#) [parity](#)

## USART FreeRTOS Driver

- *Parity setting.*  
• `usart_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.*

### 9.5.2.2 struct usart\_rtos\_handle\_t

#### Data Fields

- `USART_Type` \* `base`  
*USART base address.*
- `usart_transfer_t` `txTransfer`  
*TX transfer structure.*
- `usart_transfer_t` `rxTransfer`  
*RX transfer structure.*
- `SemaphoreHandle_t` `rxSemaphore`  
*RX semaphore for resource sharing.*
- `SemaphoreHandle_t` `txSemaphore`  
*TX semaphore for resource sharing.*
- `EventGroupHandle_t` `rxEvent`  
*RX completion event.*
- `EventGroupHandle_t` `txEvent`  
*TX completion event.*
- `void` \* `t_state`  
*Transactional state of the underlying driver.*

### 9.5.3 Function Documentation

#### 9.5.3.1 int USART\_RTOS\_Init ( usart\_rtos\_handle\_t \* handle, usart\_handle\_t \* t\_handle, const struct rtos\_usart\_config \* cfg )

##### Parameters

<i>handle</i>	The RTOS USART handle, the pointer to allocated space for RTOS context.
<i>t_handle</i>	The pointer to allocated space where to store transactional layer internal state.
<i>cfg</i>	The pointer to the parameters required to configure the USART after initialization.

##### Returns

0 succeed, others fail.



### 9.5.3.2 int USART\_RTOS\_Deinit ( usart\_rtos\_handle\_t \* *handle* )

This function deinitializes the USART module, sets all register values to reset value, and releases the resources.

## USART FreeRTOS Driver

### Parameters

<i>handle</i>	The RTOS USART handle.
---------------	------------------------

### 9.5.3.3 int USART\_RTOS\_Send ( usart\_rtos\_handle\_t \* *handle*, const uint8\_t \* *buffer*, uint32\_t *length* )

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

### Parameters

<i>handle</i>	The RTOS USART handle.
<i>buffer</i>	The pointer to buffer to send.
<i>length</i>	The number of bytes to send.

### 9.5.3.4 int USART\_RTOS\_Receive ( usart\_rtos\_handle\_t \* *handle*, uint8\_t \* *buffer*, uint32\_t *length*, size\_t \* *received* )

This function receives data from USART. 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 USART handle.
<i>buffer</i>	The pointer to buffer where to write received data.
<i>length</i>	The number of bytes to receive.
<i>received</i>	The pointer to a variable of size_t where the number of received data is filled.

## Chapter 10

# DMA: Direct Memory Access Controller Driver

### 10.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Direct Memory Access (DMA) of MCU-Xpresso SDK devices.

### 10.2 Typical use case

#### 10.2.1 DMA Operation

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

### Files

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

### Data Structures

- struct [dma\\_descriptor\\_t](#)  
*DMA descriptor structure. [More...](#)*
- struct [dma\\_xfercfg\\_t](#)  
*DMA transfer configuration. [More...](#)*
- struct [dma\\_channel\\_trigger\\_t](#)  
*DMA channel trigger. [More...](#)*
- struct [dma\\_transfer\\_config\\_t](#)  
*DMA transfer configuration. [More...](#)*
- struct [dma\\_handle\\_t](#)  
*DMA transfer handle structure. [More...](#)*

### Typedefs

- typedef void(\* [dma\\_callback](#) )(struct \_dma\_handle \*handle, void \*userData, bool transferDone, uint32\_t intmode)  
*Define Callback function for DMA.*

### Enumerations

- enum `dma_priority_t` {  
    `kDMA_ChannelPriority0` = 0,  
    `kDMA_ChannelPriority1`,  
    `kDMA_ChannelPriority2`,  
    `kDMA_ChannelPriority3`,  
    `kDMA_ChannelPriority4`,  
    `kDMA_ChannelPriority5`,  
    `kDMA_ChannelPriority6`,  
    `kDMA_ChannelPriority7` }  
    *DMA channel priority.*
- enum `dma_irq_t` {  
    `kDMA_IntA`,  
    `kDMA_IntB`,  
    `kDMA_IntError` }  
    *DMA interrupt flags.*
- enum `dma_trigger_type_t` {  
    `kDMA_NoTrigger` = 0,  
    `kDMA_LowLevelTrigger` = `DMA_CHANNEL_CFG_HWTRIGEN(1) | DMA_CHANNEL_CFG-_TRIGTYPE(1)`,  
    `kDMA_HighLevelTrigger`,  
    `kDMA_FallingEdgeTrigger` = `DMA_CHANNEL_CFG_HWTRIGEN(1)`,  
    `kDMA_RisingEdgeTrigger` }  
    *DMA trigger type.*
- enum `dma_trigger_burst_t` {  
    `kDMA_SingleTransfer` = 0,  
    `kDMA_LevelBurstTransfer` = `DMA_CHANNEL_CFG_TRIGBURST(1)`,  
    `kDMA_EdgeBurstTransfer1` = `DMA_CHANNEL_CFG_TRIGBURST(1)`,  
    `kDMA_EdgeBurstTransfer2`,  
    `kDMA_EdgeBurstTransfer4`,  
    `kDMA_EdgeBurstTransfer8`,  
    `kDMA_EdgeBurstTransfer16`,  
    `kDMA_EdgeBurstTransfer32`,  
    `kDMA_EdgeBurstTransfer64`,  
    `kDMA_EdgeBurstTransfer128`,  
    `kDMA_EdgeBurstTransfer256`,  
    `kDMA_EdgeBurstTransfer512`,  
    `kDMA_EdgeBurstTransfer1024` }  
    *DMA trigger burst.*
- enum `dma_burst_wrap_t` {  
    `kDMA_NoWrap` = 0,  
    `kDMA_SrcWrap` = `DMA_CHANNEL_CFG_SRCBURSTWRAP(1)`,  
    `kDMA_DstWrap` = `DMA_CHANNEL_CFG_DSTBURSTWRAP(1)`,  
    `kDMA_SrcAndDstWrap` }  
    *DMA burst wrapping.*
- enum `dma_transfer_type_t` {

```
kDMA_MemoryToMemory = 0x0U,
kDMA_PeripheralToMemory,
kDMA_MemoryToPeripheral,
kDMA_StaticToStatic }
```

*DMA transfer type.*

- enum `_dma_transfer_status` { `kStatus_DMA_Busy` = `MAKE_STATUS(kStatusGroup_DMA, 0)` }  
*DMA transfer status.*

## Driver version

- #define `FSL_DMA_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*DMA driver version.*

## DMA initialization and De-initialization

- void `DMA_Init` (`DMA_Type *base`)  
*Initializes DMA peripheral.*
- void `DMA_Deinit` (`DMA_Type *base`)  
*Deinitializes DMA peripheral.*

## DMA Channel Operation

- static bool `DMA_ChannelIsActive` (`DMA_Type *base`, `uint32_t channel`)  
*Return whether DMA channel is processing transfer.*
- static void `DMA_EnableChannelInterrupts` (`DMA_Type *base`, `uint32_t channel`)  
*Enables the interrupt source for the DMA transfer.*
- static void `DMA_DisableChannelInterrupts` (`DMA_Type *base`, `uint32_t channel`)  
*Disables the interrupt source for the DMA transfer.*
- static void `DMA_EnableChannel` (`DMA_Type *base`, `uint32_t channel`)  
*Enable DMA channel.*
- static void `DMA_DisableChannel` (`DMA_Type *base`, `uint32_t channel`)  
*Disable DMA channel.*
- static void `DMA_EnableChannelPeriphRq` (`DMA_Type *base`, `uint32_t channel`)  
*Set PERIPHREQEN of channel configuration register.*
- static void `DMA_DisableChannelPeriphRq` (`DMA_Type *base`, `uint32_t channel`)  
*Get PERIPHREQEN value of channel configuration register.*
- void `DMA_ConfigureChannelTrigger` (`DMA_Type *base`, `uint32_t channel`, `dma_channel_trigger_t *trigger`)  
*Set trigger settings of DMA channel.*
- `uint32_t` `DMA_GetRemainingBytes` (`DMA_Type *base`, `uint32_t channel`)  
*Gets the remaining bytes of the current DMA descriptor transfer.*
- static void `DMA_SetChannelPriority` (`DMA_Type *base`, `uint32_t channel`, `dma_priority_t priority`)  
*Set priority of channel configuration register.*
- static `dma_priority_t` `DMA_GetChannelPriority` (`DMA_Type *base`, `uint32_t channel`)  
*Get priority of channel configuration register.*
- void `DMA_CreateDescriptor` (`dma_descriptor_t *desc`, `dma_xfercfg_t *xfercfg`, void `*srcAddr`, void `*dstAddr`, void `*nextDesc`)  
*Create application specific DMA descriptor to be used in a chain in transfer.*

### DMA Transactional Operation

- void [DMA\\_AbortTransfer](#) ([dma\\_handle\\_t](#) \*handle)  
*Abort running transfer by handle.*
- void [DMA\\_CreateHandle](#) ([dma\\_handle\\_t](#) \*handle, [DMA\\_Type](#) \*base, [uint32\\_t](#) channel)  
*Creates the DMA handle.*
- void [DMA\\_SetCallback](#) ([dma\\_handle\\_t](#) \*handle, [dma\\_callback](#) callback, void \*userData)  
*Installs a callback function for the DMA transfer.*
- void [DMA\\_PrepareTransfer](#) ([dma\\_transfer\\_config\\_t](#) \*config, void \*srcAddr, void \*dstAddr, [uint32\\_t](#) byteWidth, [uint32\\_t](#) transferBytes, [dma\\_transfer\\_type\\_t](#) type, void \*nextDesc)  
*Prepares the DMA transfer structure.*
- [status\\_t](#) [DMA\\_SubmitTransfer](#) ([dma\\_handle\\_t](#) \*handle, [dma\\_transfer\\_config\\_t](#) \*config)  
*Submits the DMA transfer request.*
- void [DMA\\_StartTransfer](#) ([dma\\_handle\\_t](#) \*handle)  
*DMA start transfer.*
- void [DMA\\_HandleIRQ](#) (void)  
*DMA IRQ handler for descriptor transfer complete.*

## 10.3 Data Structure Documentation

### 10.3.1 struct dma\_descriptor\_t

#### Data Fields

- [uint32\\_t](#) [xfercfg](#)  
*Transfer configuration.*
- void \* [srcEndAddr](#)  
*Last source address of DMA transfer.*
- void \* [dstEndAddr](#)  
*Last destination address of DMA transfer.*
- void \* [linkToNextDesc](#)  
*Address of next DMA descriptor in chain.*

### 10.3.2 struct dma\_xfercfg\_t

#### Data Fields

- bool [valid](#)  
*Descriptor is ready to transfer.*
- bool [reload](#)  
*Reload channel configuration register after current descriptor is exhausted.*
- bool [swtrig](#)  
*Perform software trigger.*
- bool [clrtrig](#)  
*Clear trigger.*
- bool [intA](#)  
*Raises IRQ when transfer is done and set IRQA status register flag.*
- bool [intB](#)

- `uint8_t byteWidth`  
*Raises IRQ when transfer is done and set IRQB status register flag.*  
*Byte width of data to transfer.*
- `uint8_t srcInc`  
*Increment source address by 'srcInc' x 'byteWidth'.*
- `uint8_t dstInc`  
*Increment destination address by 'dstInc' x 'byteWidth'.*
- `uint16_t transferCount`  
*Number of transfers.*

#### 10.3.2.0.0.16 Field Documentation

##### 10.3.2.0.0.16.1 `bool dma_xfercfg_t::swtrig`

Transfer if fired when 'valid' is set

### 10.3.3 `struct dma_channel_trigger_t`

#### Data Fields

- `dma_trigger_type_t type`  
*Select hardware trigger as edge triggered or level triggered.*
- `dma_trigger_burst_t burst`  
*Select whether hardware triggers cause a single or burst transfer.*
- `dma_burst_wrap_t wrap`  
*Select wrap type, source wrap or dest wrap, or both.*

#### 10.3.3.0.0.17 Field Documentation

##### 10.3.3.0.0.17.1 `dma_trigger_type_t dma_channel_trigger_t::type`

##### 10.3.3.0.0.17.2 `dma_trigger_burst_t dma_channel_trigger_t::burst`

##### 10.3.3.0.0.17.3 `dma_burst_wrap_t dma_channel_trigger_t::wrap`

### 10.3.4 `struct dma_transfer_config_t`

#### Data Fields

- `uint8_t * srcAddr`  
*Source data address.*
- `uint8_t * dstAddr`  
*Destination data address.*
- `uint8_t * nextDesc`  
*Chain custom descriptor.*
- `dma_xfercfg_t xfercfg`  
*Transfer options.*
- `bool isPeriph`

## Enumeration Type Documentation

*DMA transfer is driven by peripheral.*

### 10.3.5 struct dma\_handle\_t

#### Data Fields

- [dma\\_callback](#) [callback](#)  
*Callback function.*
- void \* [userData](#)  
*Callback function parameter.*
- DMA\_Type \* [base](#)  
*DMA peripheral base address.*
- uint8\_t [channel](#)  
*DMA channel number.*

#### 10.3.5.0.0.18 Field Documentation

##### 10.3.5.0.0.18.1 dma\_callback dma\_handle\_t::callback

Invoked when transfer of descriptor with interrupt flag finishes

## 10.4 Macro Definition Documentation

### 10.4.1 #define FSL\_DMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

Version 2.0.0.

## 10.5 Typedef Documentation

### 10.5.1 typedef void(\* dma\_callback)(struct \_dma\_handle \*handle, void \*userData, bool transferDone, uint32\_t intmode)

## 10.6 Enumeration Type Documentation

### 10.6.1 enum dma\_priority\_t

Enumerator

- kDMA\_ChannelPriority0*** Highest channel priority - priority 0.
- kDMA\_ChannelPriority1*** Channel priority 1.
- kDMA\_ChannelPriority2*** Channel priority 2.
- kDMA\_ChannelPriority3*** Channel priority 3.
- kDMA\_ChannelPriority4*** Channel priority 4.
- kDMA\_ChannelPriority5*** Channel priority 5.
- kDMA\_ChannelPriority6*** Channel priority 6.
- kDMA\_ChannelPriority7*** Lowest channel priority - priority 7.



### 10.6.2 enum dma\_irq\_t

Enumerator

***kDMA\_IntA*** DMA interrupt flag A.  
***kDMA\_IntB*** DMA interrupt flag B.  
***kDMA\_IntError*** DMA interrupt flag error.

### 10.6.3 enum dma\_trigger\_type\_t

Enumerator

***kDMA\_NoTrigger*** Trigger is disabled.  
***kDMA\_LowLevelTrigger*** Low level active trigger.  
***kDMA\_HighLevelTrigger*** High level active trigger.  
***kDMA\_FallingEdgeTrigger*** Falling edge active trigger.  
***kDMA\_RisingEdgeTrigger*** Rising edge active trigger.

### 10.6.4 enum dma\_trigger\_burst\_t

Enumerator

***kDMA\_SingleTransfer*** Single transfer.  
***kDMA\_LevelBurstTransfer*** Burst transfer driven by level trigger.  
***kDMA\_EdgeBurstTransfer1*** Perform 1 transfer by edge trigger.  
***kDMA\_EdgeBurstTransfer2*** Perform 2 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer4*** Perform 4 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer8*** Perform 8 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer16*** Perform 16 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer32*** Perform 32 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer64*** Perform 64 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer128*** Perform 128 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer256*** Perform 256 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer512*** Perform 512 transfers by edge trigger.  
***kDMA\_EdgeBurstTransfer1024*** Perform 1024 transfers by edge trigger.

### 10.6.5 enum dma\_burst\_wrap\_t

Enumerator

***kDMA\_NoWrap*** Wrapping is disabled.  
***kDMA\_SrcWrap*** Wrapping is enabled for source.

## Function Documentation

***kDMA\_DstWrap*** Wrapping is enabled for destination.

***kDMA\_SrcAndDstWrap*** Wrapping is enabled for source and destination.

### 10.6.6 enum dma\_transfer\_type\_t

Enumerator

***kDMA\_MemoryToMemory*** Transfer from memory to memory (increment source and destination)

***kDMA\_PeripheralToMemory*** Transfer from peripheral to memory (increment only destination)

***kDMA\_MemoryToPeripheral*** Transfer from memory to peripheral (increment only source)

***kDMA\_StaticToStatic*** Peripheral to static memory (do not increment source or destination)

### 10.6.7 enum \_dma\_transfer\_status

Enumerator

***kStatus\_DMA\_Busy*** Channel is busy and can't handle the transfer request.

## 10.7 Function Documentation

### 10.7.1 void DMA\_Init ( DMA\_Type \* *base* )

This function enable the DMA clock, set descriptor table and enable DMA peripheral.

Parameters

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

### 10.7.2 void DMA\_Deinit ( DMA\_Type \* *base* )

This function gates the DMA clock.

Parameters

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

### 10.7.3 static bool DMA\_ChannelsActive ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

## Parameters

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

## Returns

True for active state, false otherwise.

#### 10.7.4 static void DMA\_EnableChannelInterrupts ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

## Parameters

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

#### 10.7.5 static void DMA\_DisableChannelInterrupts ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

## Parameters

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

#### 10.7.6 static void DMA\_EnableChannel ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

## Parameters

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

#### 10.7.7 static void DMA\_DisableChannel ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

## Function Documentation

### Parameters

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

### 10.7.8 static void DMA\_EnableChannelPeriphRq ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

### Parameters

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

### 10.7.9 static void DMA\_DisableChannelPeriphRq ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

### Parameters

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

### Returns

True for enabled PeriphRq, false for disabled.

### 10.7.10 void DMA\_ConfigureChannelTrigger ( DMA\_Type \* *base*, uint32\_t *channel*, dma\_channel\_trigger\_t \* *trigger* )

### Parameters

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

<i>trigger</i>	trigger configuration.
----------------	------------------------

#### 10.7.11 **uint32\_t DMA\_GetRemainingBytes ( DMA\_Type \* *base*, uint32\_t *channel* )**

Parameters

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

Returns

The number of bytes which have not been transferred yet.

#### 10.7.12 **static void DMA\_SetChannelPriority ( DMA\_Type \* *base*, uint32\_t *channel*, dma\_priority\_t *priority* ) [inline], [static]**

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>priority</i>	Channel priority value.

#### 10.7.13 **static dma\_priority\_t DMA\_GetChannelPriority ( DMA\_Type \* *base*, uint32\_t *channel* ) [inline], [static]**

Parameters

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

Returns

Channel priority value.

#### 10.7.14 **void DMA\_CreateDescriptor ( dma\_descriptor\_t \* *desc*, dma\_xfercfg\_t \* *xfercfg*, void \* *srcAddr*, void \* *dstAddr*, void \* *nextDesc* )**

## Function Documentation

### Parameters

<i>desc</i>	DMA descriptor address.
<i>xfercfg</i>	Transfer configuration for DMA descriptor.
<i>srcAddr</i>	Address of last item to transmit
<i>dstAddr</i>	Address of last item to receive.
<i>nextDesc</i>	Address of next descriptor in chain.

### 10.7.15 void DMA\_AbortTransfer ( dma\_handle\_t \* *handle* )

This function aborts DMA transfer specified by handle.

### Parameters

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

### 10.7.16 void DMA\_CreateHandle ( dma\_handle\_t \* *handle*, DMA\_Type \* *base*, uint32\_t *channel* )

This function is called if using transaction API for DMA. This function initializes the internal state of DMA handle.

### Parameters

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

### 10.7.17 void DMA\_SetCallback ( dma\_handle\_t \* *handle*, dma\_callback *callback*, void \* *userData* )

This callback is called in DMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

## Parameters

<i>handle</i>	DMA handle pointer.
<i>callback</i>	DMA callback function pointer.
<i>userData</i>	Parameter for callback function.

**10.7.18 void DMA\_PrepareTransfer ( dma\_transfer\_config\_t \* *config*, void \* *srcAddr*, void \* *dstAddr*, uint32\_t *byteWidth*, uint32\_t *transferBytes*, dma\_transfer\_type\_t *type*, void \* *nextDesc* )**

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

## Parameters

<i>config</i>	The user configuration structure of type dma_transfer_t.
<i>srcAddr</i>	DMA transfer source address.
<i>dstAddr</i>	DMA transfer destination address.
<i>byteWidth</i>	DMA transfer destination address width(bytes).
<i>transferBytes</i>	DMA transfer bytes to be transferred.
<i>type</i>	DMA transfer type.
<i>nextDesc</i>	Chain custom descriptor to transfer.

## Note

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

**10.7.19 status\_t DMA\_SubmitTransfer ( dma\_handle\_t \* *handle*, dma\_transfer\_config\_t \* *config* )**

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

## Function Documentation

### Parameters

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

### Return values

<i>kStatus_DMA_Success</i>	It means submit transfer request succeed.
<i>kStatus_DMA_QueueFull</i>	It means TCD queue is full. Submit transfer request is not allowed.
<i>kStatus_DMA_Busy</i>	It means the given channel is busy, need to submit request later.

### 10.7.20 void DMA\_StartTransfer ( dma\_handle\_t \* *handle* )

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

### Parameters

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

### 10.7.21 void DMA\_HandleIRQ ( void )

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



## Chapter 11

# SCTimer: SCTimer/PWM (SCT)

### 11.1 Overview

The MCUXpresso SDK provides a driver for the SCTimer Module (SCT) of MCUXpresso SDK devices.

### 11.2 Function groups

The SCTimer driver supports the generation of PWM signals. The driver also supports enabling events in various states of the SCTimer and the actions that will be triggered when an event occurs.

#### 11.2.1 Initialization and deinitialization

The function [SCTIMER\\_Init\(\)](#) initializes the SCTimer with specified configurations. The function [SCTIMER\\_GetDefaultConfig\(\)](#) gets the default configurations.

The function [SCTIMER\\_Deinit\(\)](#) halts the SCTimer counter and turns off the module clock.

#### 11.2.2 PWM Operations

The function [SCTIMER\\_SetupPwm\(\)](#) sets up SCTimer channels for PWM output. The function can set up the PWM signal properties duty cycle and level-mode (active low or high) to use. However, the same PWM period and PWM mode (edge or center-aligned) 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 1 and 100.

The function [SCTIMER\\_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular SCTimer channel.

#### 11.2.3 Status

Provides functions to get and clear the SCTimer status.

#### 11.2.4 Interrupt

Provides functions to enable/disable SCTimer interrupts and get current enabled interrupts.

### 11.3 SCTimer State machine and operations

The SCTimer has 10 states and each state can have a set of events enabled that can trigger a user specified action when the event occurs.

#### 11.3.1 SCTimer event operations

The user can create an event and enable it in the current state using the functions [SCTIMER\\_CreateAndScheduleEvent\(\)](#) and [SCTIMER\\_ScheduleEvent\(\)](#). [SCTIMER\\_CreateAndScheduleEvent\(\)](#) creates a new event based on the users preference and enables it in the current state. [SCTIMER\\_ScheduleEvent\(\)](#) enables an event created earlier in the current state.

#### 11.3.2 SCTimer state operations

The user can get the current state number by calling [SCTIMER\\_GetCurrentState\(\)](#), he can use this state number to set state transitions when a particular event is triggered.

Once the user has created and enabled events for the current state he can go to the next state by calling the function [SCTIMER\\_IncreaseState\(\)](#). The user can then start creating events to be enabled in this new state.

#### 11.3.3 SCTimer action operations

There are a set of functions that decide what action should be taken when an event is triggered. [SCTIMER\\_SetupCaptureAction\(\)](#) sets up which counter to capture and which capture register to read on event trigger. [SCTIMER\\_SetupNextStateAction\(\)](#) sets up which state the SCTimer state machine should transition to on event trigger. [SCTIMER\\_SetupOutputSetAction\(\)](#) sets up which pin to set on event trigger. [SCTIMER\\_SetupOutputClearAction\(\)](#) sets up which pin to clear on event trigger. [SCTIMER\\_SetupOutputToggleAction\(\)](#) sets up which pin to toggle on event trigger. [SCTIMER\\_SetupCounterLimitAction\(\)](#) sets up which counter will be limited on event trigger. [SCTIMER\\_SetupCounterStopAction\(\)](#) sets up which counter will be stopped on event trigger. [SCTIMER\\_SetupCounterStartAction\(\)](#) sets up which counter will be started on event trigger. [SCTIMER\\_SetupCounterHaltAction\(\)](#) sets up which counter will be halted on event trigger. [SCTIMER\\_SetupDmaTriggerAction\(\)](#) sets up which DMA request will be activated on event trigger.

### 11.4 16-bit counter mode

The SCTimer is configurable to run as two 16-bit counters via the enableCounterUnify flag that is available in the configuration structure passed in to the [SCTIMER\\_Init\(\)](#) function.

When operating in 16-bit mode, it is important the user specify the appropriate counter to use when working with the functions: [SCTIMER\\_StartTimer\(\)](#), [SCTIMER\\_StopTimer\(\)](#), [SCTIMER\\_CreateAndScheduleEvent\(\)](#), [SCTIMER\\_SetupCaptureAction\(\)](#), [SCTIMER\\_SetupCounterLimitAction\(\)](#), [SCTIM-](#)

[ER\\_SetupCounterStopAction\(\)](#), [SCTIMER\\_SetupCounterStartAction\(\)](#), [SCTIMER\\_SetupCounterHaltAction\(\)](#).

## 11.5 Typical use case

### 11.5.1 PWM output

Output a PWM signal on 2 SCTimer channels with different duty cycles. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/sctimer

## Files

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

## Data Structures

- struct [sctimer\\_pwm\\_signal\\_param\\_t](#)  
*Options to configure a SCTimer PWM signal. [More...](#)*
- struct [sctimer\\_config\\_t](#)  
*SCTimer configuration structure. [More...](#)*

## Typedefs

- typedef void(\* [sctimer\\_event\\_callback\\_t](#))(void)  
*SCTimer callback typedef.*

## Enumerations

- enum [sctimer\\_pwm\\_mode\\_t](#) {  
    [kSCTIMER\\_EdgeAlignedPwm](#) = 0U,  
    [kSCTIMER\\_CenterAlignedPwm](#) }  
*SCTimer PWM operation modes.*
- enum [sctimer\\_counter\\_t](#) {  
    [kSCTIMER\\_Counter\\_L](#) = 0U,  
    [kSCTIMER\\_Counter\\_H](#) }  
*SCTimer counters when working as two independent 16-bit counters.*
- enum [sctimer\\_input\\_t](#) {  
    [kSCTIMER\\_Input\\_0](#) = 0U,  
    [kSCTIMER\\_Input\\_1](#),  
    [kSCTIMER\\_Input\\_2](#),  
    [kSCTIMER\\_Input\\_3](#),  
    [kSCTIMER\\_Input\\_4](#),  
    [kSCTIMER\\_Input\\_5](#),  
    [kSCTIMER\\_Input\\_6](#),  
    [kSCTIMER\\_Input\\_7](#) }  
*List of SCTimer input pins.*

## Typical use case

- enum `sctimer_out_t` {  
    `kSCTIMER_Out_0` = 0U,  
    `kSCTIMER_Out_1`,  
    `kSCTIMER_Out_2`,  
    `kSCTIMER_Out_3`,  
    `kSCTIMER_Out_4`,  
    `kSCTIMER_Out_5`,  
    `kSCTIMER_Out_6`,  
    `kSCTIMER_Out_7` }  
    *List of SCTimer output pins.*
- enum `sctimer_pwm_level_select_t` {  
    `kSCTIMER_LowTrue` = 0U,  
    `kSCTIMER_HighTrue` }  
    *SCTimer PWM output pulse mode: high-true, low-true or no output.*
- enum `sctimer_clock_mode_t` {  
    `kSCTIMER_System_ClockMode` = 0U,  
    `kSCTIMER_Sampled_ClockMode`,  
    `kSCTIMER_Input_ClockMode`,  
    `kSCTIMER_Asynchronous_ClockMode` }  
    *SCTimer clock mode options.*
- enum `sctimer_clock_select_t` {  
    `kSCTIMER_Clock_On_Rise_Input_0` = 0U,  
    `kSCTIMER_Clock_On_Fall_Input_0`,  
    `kSCTIMER_Clock_On_Rise_Input_1`,  
    `kSCTIMER_Clock_On_Fall_Input_1`,  
    `kSCTIMER_Clock_On_Rise_Input_2`,  
    `kSCTIMER_Clock_On_Fall_Input_2`,  
    `kSCTIMER_Clock_On_Rise_Input_3`,  
    `kSCTIMER_Clock_On_Fall_Input_3`,  
    `kSCTIMER_Clock_On_Rise_Input_4`,  
    `kSCTIMER_Clock_On_Fall_Input_4`,  
    `kSCTIMER_Clock_On_Rise_Input_5`,  
    `kSCTIMER_Clock_On_Fall_Input_5`,  
    `kSCTIMER_Clock_On_Rise_Input_6`,  
    `kSCTIMER_Clock_On_Fall_Input_6`,  
    `kSCTIMER_Clock_On_Rise_Input_7`,  
    `kSCTIMER_Clock_On_Fall_Input_7` }  
    *SCTimer clock select options.*
- enum `sctimer_conflict_resolution_t` {  
    `kSCTIMER_ResolveNone` = 0U,  
    `kSCTIMER_ResolveSet`,  
    `kSCTIMER_ResolveClear`,  
    `kSCTIMER_ResolveToggle` }  
    *SCTimer output conflict resolution options.*
- enum `sctimer_event_t`  
    *List of SCTimer event types.*

- enum `sctimer_interrupt_enable_t` {  
`kSCTIMER_Event0InterruptEnable` = (1U << 0),  
`kSCTIMER_Event1InterruptEnable` = (1U << 1),  
`kSCTIMER_Event2InterruptEnable` = (1U << 2),  
`kSCTIMER_Event3InterruptEnable` = (1U << 3),  
`kSCTIMER_Event4InterruptEnable` = (1U << 4),  
`kSCTIMER_Event5InterruptEnable` = (1U << 5),  
`kSCTIMER_Event6InterruptEnable` = (1U << 6),  
`kSCTIMER_Event7InterruptEnable` = (1U << 7),  
`kSCTIMER_Event8InterruptEnable` = (1U << 8),  
`kSCTIMER_Event9InterruptEnable` = (1U << 9),  
`kSCTIMER_Event10InterruptEnable` = (1U << 10),  
`kSCTIMER_Event11InterruptEnable` = (1U << 11),  
`kSCTIMER_Event12InterruptEnable` = (1U << 12) }

*List of SCTimer interrupts.*

- enum `sctimer_status_flags_t` {  
`kSCTIMER_Event0Flag` = (1U << 0),  
`kSCTIMER_Event1Flag` = (1U << 1),  
`kSCTIMER_Event2Flag` = (1U << 2),  
`kSCTIMER_Event3Flag` = (1U << 3),  
`kSCTIMER_Event4Flag` = (1U << 4),  
`kSCTIMER_Event5Flag` = (1U << 5),  
`kSCTIMER_Event6Flag` = (1U << 6),  
`kSCTIMER_Event7Flag` = (1U << 7),  
`kSCTIMER_Event8Flag` = (1U << 8),  
`kSCTIMER_Event9Flag` = (1U << 9),  
`kSCTIMER_Event10Flag` = (1U << 10),  
`kSCTIMER_Event11Flag` = (1U << 11),  
`kSCTIMER_Event12Flag` = (1U << 12),  
`kSCTIMER_BusErrorLFlag`,  
`kSCTIMER_BusErrorHFlag` }

*List of SCTimer flags.*

## Driver version

- #define `FSL_SCTIMER_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*Version 2.0.0.*

## Initialization and deinitialization

- `status_t` `SCTIMER_Init` (`SCT_Type *base`, `const sctimer_config_t *config`)  
*Ungates the SCTimer clock and configures the peripheral for basic operation.*
- void `SCTIMER_Deinit` (`SCT_Type *base`)  
*Gates the SCTimer clock.*
- void `SCTIMER_GetDefaultConfig` (`sctimer_config_t *config`)  
*Fills in the SCTimer configuration structure with the default settings.*

## Typical use case

### PWM setup operations

- [status\\_t SCTIMER\\_SetupPwm](#) (SCT\_Type \*base, const [sctimer\\_pwm\\_signal\\_param\\_t](#) \*pwmParams, [sctimer\\_pwm\\_mode\\_t](#) mode, uint32\_t pwmFreq\_Hz, uint32\_t srcClock\_Hz, uint32\_t \*event)  
*Configures the PWM signal parameters.*
- void [SCTIMER\\_UpdatePwmDutycycle](#) (SCT\_Type \*base, [sctimer\\_out\\_t](#) output, uint8\_t dutyCyclePercent, uint32\_t event)  
*Updates the duty cycle of an active PWM signal.*

### Interrupt Interface

- static void [SCTIMER\\_EnableInterrupts](#) (SCT\_Type \*base, uint32\_t mask)  
*Enables the selected SCTimer interrupts.*
- static void [SCTIMER\\_DisableInterrupts](#) (SCT\_Type \*base, uint32\_t mask)  
*Disables the selected SCTimer interrupts.*
- static uint32\_t [SCTIMER\\_GetEnabledInterrupts](#) (SCT\_Type \*base)  
*Gets the enabled SCTimer interrupts.*

### Status Interface

- static uint32\_t [SCTIMER\\_GetStatusFlags](#) (SCT\_Type \*base)  
*Gets the SCTimer status flags.*
- static void [SCTIMER\\_ClearStatusFlags](#) (SCT\_Type \*base, uint32\_t mask)  
*Clears the SCTimer status flags.*

### Counter Start and Stop

- static void [SCTIMER\\_StartTimer](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) countertoStart)  
*Starts the SCTimer counter.*
- static void [SCTIMER\\_StopTimer](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) countertoStop)  
*Halts the SCTimer counter.*

### Functions to create a new event and manage the state logic

- [status\\_t SCTIMER\\_CreateAndScheduleEvent](#) (SCT\_Type \*base, [sctimer\\_event\\_t](#) howToMonitor, uint32\_t matchValue, uint32\_t whichIO, [sctimer\\_counter\\_t](#) whichCounter, uint32\_t \*event)  
*Create an event that is triggered on a match or IO and schedule in current state.*
- void [SCTIMER\\_ScheduleEvent](#) (SCT\_Type \*base, uint32\_t event)  
*Enable an event in the current state.*
- [status\\_t SCTIMER\\_IncreaseState](#) (SCT\_Type \*base)  
*Increase the state by 1.*
- uint32\_t [SCTIMER\\_GetCurrentState](#) (SCT\_Type \*base)  
*Provides the current state.*

### Actions to take in response to an event

- [status\\_t SCTIMER\\_SetupCaptureAction](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) whichCounter, uint32\_t \*captureRegister, uint32\_t event)  
*Setup capture of the counter value on trigger of a selected event.*

- void [SCTIMER\\_SetCallback](#) (SCT\_Type \*base, [sctimer\\_event\\_callback\\_t](#) callback, uint32\_t event)  
*Receive notification when the event trigger an interrupt.*
- static void [SCTIMER\\_SetupNextStateAction](#) (SCT\_Type \*base, uint32\_t nextState, uint32\_t event)  
*Transition to the specified state.*
- static void [SCTIMER\\_SetupOutputSetAction](#) (SCT\_Type \*base, uint32\_t whichIO, uint32\_t event)  
*Set the Output.*
- static void [SCTIMER\\_SetupOutputClearAction](#) (SCT\_Type \*base, uint32\_t whichIO, uint32\_t event)  
*Clear the Output.*
- void [SCTIMER\\_SetupOutputToggleAction](#) (SCT\_Type \*base, uint32\_t whichIO, uint32\_t event)  
*Toggle the output level.*
- static void [SCTIMER\\_SetupCounterLimitAction](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) whichCounter, uint32\_t event)  
*Limit the running counter.*
- static void [SCTIMER\\_SetupCounterStopAction](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) whichCounter, uint32\_t event)  
*Stop the running counter.*
- static void [SCTIMER\\_SetupCounterStartAction](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) whichCounter, uint32\_t event)  
*Re-start the stopped counter.*
- static void [SCTIMER\\_SetupCounterHaltAction](#) (SCT\_Type \*base, [sctimer\\_counter\\_t](#) whichCounter, uint32\_t event)  
*Halt the running counter.*
- static void [SCTIMER\\_SetupDmaTriggerAction](#) (SCT\_Type \*base, uint32\_t dmaNumber, uint32\_t event)  
*Generate a DMA request.*
- void [SCTIMER\\_EventHandleIRQ](#) (SCT\_Type \*base)  
*SCTimer interrupt handler.*

## 11.6 Data Structure Documentation

### 11.6.1 struct sctimer\_pwm\_signal\_param\_t

#### Data Fields

- [sctimer\\_out\\_t](#) output  
*The output pin to use to generate the PWM signal.*
- [sctimer\\_pwm\\_level\\_select\\_t](#) level  
*PWM output active level select.*
- uint8\_t [dutyCyclePercent](#)  
*PWM pulse width, value should be between 1 to 100 100 = always active signal (100% duty cycle).*

## Typedef Documentation

### 11.6.1.0.0.19 Field Documentation

11.6.1.0.0.19.1 `sctimer_pwm_level_select_t sctimer_pwm_signal_param_t::level`

11.6.1.0.0.19.2 `uint8_t sctimer_pwm_signal_param_t::dutyCyclePercent`

### 11.6.2 struct sctimer\_config\_t

This structure holds the configuration settings for the SCTimer peripheral. To initialize this structure to reasonable defaults, call the `SCTMR_GetDefaultConfig()` function and pass a pointer to the configuration structure instance.

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

## Data Fields

- bool `enableCounterUnify`  
*true: SCT operates as a unified 32-bit counter; false: SCT operates as two 16-bit counters*
- `sctimer_clock_mode_t clockMode`  
*SCT clock mode value.*
- `sctimer_clock_select_t clockSelect`  
*SCT clock select value.*
- bool `enableBidirection_l`  
*true: Up-down count mode for the L or unified counter false: Up count mode only for the L or unified counter*
- bool `enableBidirection_h`  
*true: Up-down count mode for the H or unified counter false: Up count mode only for the H or unified counter.*
- `uint8_t prescale_l`  
*Prescale value to produce the L or unified counter clock.*
- `uint8_t prescale_h`  
*Prescale value to produce the H counter clock.*
- `uint8_t outInitState`  
*Defines the initial output value.*

### 11.6.2.0.0.20 Field Documentation

11.6.2.0.0.20.1 `bool sctimer_config_t::enableBidirection_h`

This field is used only if the `enableCounterUnify` is set to false

11.6.2.0.0.20.2 `uint8_t sctimer_config_t::prescale_h`

This field is used only if the `enableCounterUnify` is set to false

## 11.7 Typedef Documentation

11.7.1 `typedef void(* sctimer_event_callback_t)(void)`



## 11.8 Enumeration Type Documentation

### 11.8.1 enum sctimer\_pwm\_mode\_t

Enumerator

*kSCTIMER\_EdgeAlignedPwm* Edge-aligned PWM.  
*kSCTIMER\_CenterAlignedPwm* Center-aligned PWM.

### 11.8.2 enum sctimer\_counter\_t

Enumerator

*kSCTIMER\_Counter\_L* Counter L.  
*kSCTIMER\_Counter\_H* Counter H.

### 11.8.3 enum sctimer\_input\_t

Enumerator

*kSCTIMER\_Input\_0* SCTIMER input 0.  
*kSCTIMER\_Input\_1* SCTIMER input 1.  
*kSCTIMER\_Input\_2* SCTIMER input 2.  
*kSCTIMER\_Input\_3* SCTIMER input 3.  
*kSCTIMER\_Input\_4* SCTIMER input 4.  
*kSCTIMER\_Input\_5* SCTIMER input 5.  
*kSCTIMER\_Input\_6* SCTIMER input 6.  
*kSCTIMER\_Input\_7* SCTIMER input 7.

### 11.8.4 enum sctimer\_out\_t

Enumerator

*kSCTIMER\_Out\_0* SCTIMER output 0.  
*kSCTIMER\_Out\_1* SCTIMER output 1.  
*kSCTIMER\_Out\_2* SCTIMER output 2.  
*kSCTIMER\_Out\_3* SCTIMER output 3.  
*kSCTIMER\_Out\_4* SCTIMER output 4.  
*kSCTIMER\_Out\_5* SCTIMER output 5.  
*kSCTIMER\_Out\_6* SCTIMER output 6.  
*kSCTIMER\_Out\_7* SCTIMER output 7.

### 11.8.5 enum sctimer\_pwm\_level\_select\_t

Enumerator

*kSCTIMER\_LowTrue* Low true pulses.  
*kSCTIMER\_HighTrue* High true pulses.

### 11.8.6 enum sctimer\_clock\_mode\_t

Enumerator

*kSCTIMER\_System\_ClockMode* System Clock Mode.  
*kSCTIMER\_Sampled\_ClockMode* Sampled System Clock Mode.  
*kSCTIMER\_Input\_ClockMode* SCT Input Clock Mode.  
*kSCTIMER\_Asynchronous\_ClockMode* Asynchronous Mode.

### 11.8.7 enum sctimer\_clock\_select\_t

Enumerator

*kSCTIMER\_Clock\_On\_Rise\_Input\_0* Rising edges on input 0.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_0* Falling edges on input 0.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_1* Rising edges on input 1.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_1* Falling edges on input 1.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_2* Rising edges on input 2.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_2* Falling edges on input 2.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_3* Rising edges on input 3.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_3* Falling edges on input 3.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_4* Rising edges on input 4.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_4* Falling edges on input 4.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_5* Rising edges on input 5.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_5* Falling edges on input 5.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_6* Rising edges on input 6.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_6* Falling edges on input 6.  
*kSCTIMER\_Clock\_On\_Rise\_Input\_7* Rising edges on input 7.  
*kSCTIMER\_Clock\_On\_Fall\_Input\_7* Falling edges on input 7.

### 11.8.8 enum sctimer\_conflict\_resolution\_t

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

## Enumerator

***kSCTIMER\_ResolveNone*** No change.  
***kSCTIMER\_ResolveSet*** Set output.  
***kSCTIMER\_ResolveClear*** Clear output.  
***kSCTIMER\_ResolveToggle*** Toggle output.

### 11.8.9 enum sctimer\_interrupt\_enable\_t

## Enumerator

***kSCTIMER\_Event0InterruptEnable*** Event 0 interrupt.  
***kSCTIMER\_Event1InterruptEnable*** Event 1 interrupt.  
***kSCTIMER\_Event2InterruptEnable*** Event 2 interrupt.  
***kSCTIMER\_Event3InterruptEnable*** Event 3 interrupt.  
***kSCTIMER\_Event4InterruptEnable*** Event 4 interrupt.  
***kSCTIMER\_Event5InterruptEnable*** Event 5 interrupt.  
***kSCTIMER\_Event6InterruptEnable*** Event 6 interrupt.  
***kSCTIMER\_Event7InterruptEnable*** Event 7 interrupt.  
***kSCTIMER\_Event8InterruptEnable*** Event 8 interrupt.  
***kSCTIMER\_Event9InterruptEnable*** Event 9 interrupt.  
***kSCTIMER\_Event10InterruptEnable*** Event 10 interrupt.  
***kSCTIMER\_Event11InterruptEnable*** Event 11 interrupt.  
***kSCTIMER\_Event12InterruptEnable*** Event 12 interrupt.

### 11.8.10 enum sctimer\_status\_flags\_t

## Enumerator

***kSCTIMER\_Event0Flag*** Event 0 Flag.  
***kSCTIMER\_Event1Flag*** Event 1 Flag.  
***kSCTIMER\_Event2Flag*** Event 2 Flag.  
***kSCTIMER\_Event3Flag*** Event 3 Flag.  
***kSCTIMER\_Event4Flag*** Event 4 Flag.  
***kSCTIMER\_Event5Flag*** Event 5 Flag.  
***kSCTIMER\_Event6Flag*** Event 6 Flag.  
***kSCTIMER\_Event7Flag*** Event 7 Flag.  
***kSCTIMER\_Event8Flag*** Event 8 Flag.  
***kSCTIMER\_Event9Flag*** Event 9 Flag.  
***kSCTIMER\_Event10Flag*** Event 10 Flag.  
***kSCTIMER\_Event11Flag*** Event 11 Flag.  
***kSCTIMER\_Event12Flag*** Event 12 Flag.  
***kSCTIMER\_BusErrorLFlag*** Bus error due to write when L counter was not halted.  
***kSCTIMER\_BusErrorHFlag*** Bus error due to write when H counter was not halted.

## Function Documentation

### 11.9 Function Documentation

#### 11.9.1 **status\_t SCTIMER\_Init ( SCT\_Type \* *base*, const sctimer\_config\_t \* *config* )**

Note

This API should be called at the beginning of the application using the SCTimer driver.

Parameters

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

Returns

kStatus\_Success indicates success; Else indicates failure.

#### 11.9.2 **void SCTIMER\_Deinit ( SCT\_Type \* *base* )**

Parameters

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

#### 11.9.3 **void SCTIMER\_GetDefaultConfig ( sctimer\_config\_t \* *config* )**

The default values are:

```
* config->enableCounterUnify = true;
* config->clockMode = kSCTIMER_System_ClockMode;
* config->clockSelect = kSCTIMER_Clock_On_Rise_Input_0;
* config->enableBidirection_l = false;
* config->enableBidirection_h = false;
* config->prescale_l = 0;
* config->prescale_h = 0;
* config->outInitState = 0;
*
```

Parameters

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

#### 11.9.4 **status\_t SCTIMER\_SetupPwm ( SCT\_Type \* *base*, const sctimer\_pwm\_signal\_param\_t \* *pwmParams*, sctimer\_pwm\_mode\_t *mode*, uint32\_t *pwmFreq\_Hz*, uint32\_t *srcClock\_Hz*, uint32\_t \* *event* )**

Call this function to configure the PWM signal period, mode, duty cycle, and edge. This function will create 2 events; one of the events will trigger on match with the pulse value and the other will trigger when the counter matches the PWM period. The PWM period event is also used as a limit event to reset the counter or change direction. Both events are enabled for the same state. The state number can be retrieved by calling the function SCTIMER\_GetCurrentStateNumber(). The counter is set to operate as one 32-bit counter (unify bit is set to 1). The counter operates in bi-directional mode when generating a center-aligned PWM.

#### Note

When setting PWM output from multiple output pins, they all should use the same PWM mode. For example, all PWM's should be either edge-aligned or center-aligned. When using this API, the PWM signal frequency of all the initialized channels must be the same. Otherwise all the initialized channels' PWM signal frequency is equal to the last call to the API's *pwmFreq\_Hz*.

#### Parameters

<i>base</i>	SCTimer peripheral base address
<i>pwmParams</i>	PWM parameters to configure the output
<i>mode</i>	PWM operation mode, options available in enumeration <a href="#">sctimer_pwm_mode_t</a>
<i>pwmFreq_Hz</i>	PWM signal frequency in Hz
<i>srcClock_Hz</i>	SCTimer counter clock in Hz
<i>event</i>	Pointer to a variable where the PWM period event number is stored

#### Returns

kStatus\_Success on success  
kStatus\_Fail If we have hit the limit in terms of number of events created or if an incorrect PWM duty cycle is passed in.

#### 11.9.5 **void SCTIMER\_UpdatePwmDutycycle ( SCT\_Type \* *base*, sctimer\_out\_t *output*, uint8\_t *dutyCyclePercent*, uint32\_t *event* )**

## Function Documentation

### Parameters

<i>base</i>	SCTimer peripheral base address
<i>output</i>	The output to configure
<i>dutyCycle-Percent</i>	New PWM pulse width; the value should be between 1 to 100
<i>event</i>	Event number associated with this PWM signal. This was returned to the user by the function <a href="#">SCTIMER_SetupPwm()</a> .

### 11.9.6 static void SCTIMER\_EnableInterrupts ( SCT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

### Parameters

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

### 11.9.7 static void SCTIMER\_DisableInterrupts ( SCT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

### Parameters

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

### 11.9.8 static uint32\_t SCTIMER\_GetEnabledInterrupts ( SCT\_Type \* *base* ) [inline], [static]

### Parameters

---

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [sctimer\\_interrupt\\_enable\\_t](#)

#### 11.9.9 static uint32\_t SCTIMER\_GetStatusFlags ( SCT\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [sctimer\\_status\\_flags\\_t](#)

#### 11.9.10 static void SCTIMER\_ClearStatusFlags ( SCT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

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

#### 11.9.11 static void SCTIMER\_StartTimer ( SCT\_Type \* *base*, sctimer\_counter\_t *countertoStart* ) [inline], [static]

Parameters

## Function Documentation

<i>base</i>	SCTimer peripheral base address
<i>countertoStart</i>	SCTimer counter to start; if unify mode is set then function always writes to HALT_L bit

**11.9.12 static void SCTIMER\_StopTimer ( SCT\_Type \* *base*, sctimer\_counter\_t *countertoStop* ) [inline], [static]**

### Parameters

<i>base</i>	SCTimer peripheral base address
<i>countertoStop</i>	SCTimer counter to stop; if unify mode is set then function always writes to HALT_L bit

**11.9.13 status\_t SCTIMER\_CreateAndScheduleEvent ( SCT\_Type \* *base*, sctimer\_event\_t *howToMonitor*, uint32\_t *matchValue*, uint32\_t *whichIO*, sctimer\_counter\_t *whichCounter*, uint32\_t \* *event* )**

This function will configure an event using the options provided by the user. If the event type uses the counter match, then the function will set the user provided match value into a match register and put this match register number into the event control register. The event is enabled for the current state and the event number is increased by one at the end. The function returns the event number; this event number can be used to configure actions to be done when this event is triggered.

### Parameters

<i>base</i>	SCTimer peripheral base address
<i>howToMonitor</i>	Event type; options are available in the enumeration <a href="#">sctimer_interrupt_enable_t</a>
<i>matchValue</i>	The match value that will be programmed to a match register
<i>whichIO</i>	The input or output that will be involved in event triggering. This field is ignored if the event type is "match only"
<i>whichCounter</i>	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as we have only 1 unified counter; hence ignored.



<i>event</i>	Pointer to a variable where the new event number is stored
--------------	--

## Returns

kStatus\_Success on success kStatus\_Error if we have hit the limit in terms of number of events created or if we have reached the limit in terms of number of match registers

#### 11.9.14 void SCTIMER\_ScheduleEvent ( SCT\_Type \* *base*, uint32\_t *event* )

This function will allow the event passed in to trigger in the current state. The event must be created earlier by either calling the function [SCTIMER\\_SetupPwm\(\)](#) or function [SCTIMER\\_CreateAndScheduleEvent\(\)](#).

## Parameters

<i>base</i>	SCTimer peripheral base address
<i>event</i>	Event number to enable in the current state

#### 11.9.15 status\_t SCTIMER\_IncreaseState ( SCT\_Type \* *base* )

All future events created by calling the function [SCTIMER\\_ScheduleEvent\(\)](#) will be enabled in this new state.

## Parameters

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

## Returns

kStatus\_Success on success kStatus\_Error if we have hit the limit in terms of states used

#### 11.9.16 uint32\_t SCTIMER\_GetCurrentState ( SCT\_Type \* *base* )

User can use this to set the next state by calling the function [SCTIMER\\_SetupNextStateAction\(\)](#).

## Function Documentation

### Parameters

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

### Returns

The current state

**11.9.17** `status_t SCTIMER_SetupCaptureAction ( SCT_Type * base,  
sctimer_counter_t whichCounter, uint32_t * captureRegister, uint32_t  
event )`

### Parameters

<i>base</i>	SCTimer peripheral base address
<i>whichCounter</i>	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
<i>captureRegister</i>	Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
<i>event</i>	Event number that will trigger the capture

### Returns

kStatus\_Success on success kStatus\_Error if we have hit the limit in terms of number of match/capture registers available

**11.9.18** `void SCTIMER_SetCallback ( SCT_Type * base, sctimer_event_callback_t  
callback, uint32_t event )`

If the interrupt for the event is enabled by the user, then a callback can be registered which will be invoked when the event is triggered

### Parameters

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

<i>event</i>	Event number that will trigger the interrupt
<i>callback</i>	Function to invoke when the event is triggered

#### 11.9.19 static void SCTIMER\_SetupNextStateAction ( SCT\_Type \* *base*, uint32\_t *nextState*, uint32\_t *event* ) [inline], [static]

This transition will be triggered by the event number that is passed in by the user.

Parameters

<i>base</i>	SCTimer peripheral base address
<i>nextState</i>	The next state SCTimer will transition to
<i>event</i>	Event number that will trigger the state transition

#### 11.9.20 static void SCTIMER\_SetupOutputSetAction ( SCT\_Type \* *base*, uint32\_t *whichIO*, uint32\_t *event* ) [inline], [static]

This output will be set when the event number that is passed in by the user is triggered.

Parameters

<i>base</i>	SCTimer peripheral base address
<i>whichIO</i>	The output to set
<i>event</i>	Event number that will trigger the output change

#### 11.9.21 static void SCTIMER\_SetupOutputClearAction ( SCT\_Type \* *base*, uint32\_t *whichIO*, uint32\_t *event* ) [inline], [static]

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

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

## Function Documentation

<i>whichIO</i>	The output to clear
<i>event</i>	Event number that will trigger the output change

### 11.9.22 void SCTIMER\_SetupOutputToggleAction ( SCT\_Type \* *base*, uint32\_t *whichIO*, uint32\_t *event* )

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

<i>base</i>	SCTimer peripheral base address
<i>whichIO</i>	The output to toggle
<i>event</i>	Event number that will trigger the output change

### 11.9.23 static void SCTIMER\_SetupCounterLimitAction ( SCT\_Type \* *base*, sctimer\_counter\_t *whichCounter*, uint32\_t *event* ) [inline], [static]

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

<i>base</i>	SCTimer peripheral base address
<i>whichCounter</i>	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
<i>event</i>	Event number that will trigger the counter to be limited

### 11.9.24 static void SCTIMER\_SetupCounterStopAction ( SCT\_Type \* *base*, sctimer\_counter\_t *whichCounter*, uint32\_t *event* ) [inline], [static]

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

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

<i>whichCounter</i>	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
<i>event</i>	Event number that will trigger the counter to be stopped

**11.9.25 static void SCTIMER\_SetupCounterStartAction ( SCT\_Type \* *base*,  
sctimer\_counter\_t *whichCounter*, uint32\_t *event* ) [inline], [static]**

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

<i>base</i>	SCTimer peripheral base address
<i>whichCounter</i>	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
<i>event</i>	Event number that will trigger the counter to re-start

**11.9.26 static void SCTIMER\_SetupCounterHaltAction ( SCT\_Type \* *base*,  
sctimer\_counter\_t *whichCounter*, uint32\_t *event* ) [inline], [static]**

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the [SCTIMER\\_StartTimer\(\)](#) function.

Parameters

<i>base</i>	SCTimer peripheral base address
<i>whichCounter</i>	SCTimer counter to use when operating in 16-bit mode. In 32-bit mode, this field has no meaning as only the Counter_L bits are used.
<i>event</i>	Event number that will trigger the counter to be halted

**11.9.27 static void SCTIMER\_SetupDmaTriggerAction ( SCT\_Type \* *base*, uint32\_t  
*dmaNumber*, uint32\_t *event* ) [inline], [static]**

DMA request will be triggered by the event number that is passed in by the user.

## Function Documentation

### Parameters

<i>base</i>	SCTimer peripheral base address
<i>dmaNumber</i>	The DMA request to generate
<i>event</i>	Event number that will trigger the DMA request

### 11.9.28 void SCTIMER\_EventHandleIRQ ( SCT\_Type \* *base* )

### Parameters

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

## Chapter 12

# WWDT: Windowed Watchdog Timer Driver

### 12.1 Overview

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

### 12.2 Function groups

#### 12.2.1 Initialization and deinitialization

The function [WWDT\\_Init\(\)](#) initializes the watchdog timer with specified configurations. The configurations include timeout value and whether to enable watchdog after init. The function [WWDT\\_GetDefaultConfig\(\)](#) gets the default configurations.

The function [WWDT\\_Deinit\(\)](#) disables the watchdog and the module clock.

#### 12.2.2 Status

Provides functions to get and clear the WWDT status.

#### 12.2.3 Interrupt

Provides functions to enable/disable WWDT interrupts and get current enabled interrupts.

#### 12.2.4 Watch dog Refresh

The function [WWDT\\_Refresh\(\)](#) feeds the WWDT.

### 12.3 Typical use case

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

### Files

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

### Data Structures

- struct [wwdt\\_config\\_t](#)  
*Describes WWDT configuration structure. [More...](#)*

## Typical use case

## Enumerations

- enum `_wwdt_status_flags_t` {  
    `kWWDT_TimeoutFlag` = `WWDT_MOD_WDTOF_MASK`,  
    `kWWDT_WarningFlag` = `WWDT_MOD_WDINT_MASK` }  
    *WWDT status flags.*

## Driver version

- #define `FSL_WWDT_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
    *Defines WWDT driver version 2.0.0.*

## Refresh sequence

- #define `WWDT_FIRST_WORD_OF_REFRESH` (`0xAAU`)  
    *First word of refresh sequence.*
- #define `WWDT_SECOND_WORD_OF_REFRESH` (`0x55U`)  
    *Second word of refresh sequence.*

## WWDT Initialization and De-initialization

- void `WWDT_GetDefaultConfig` (`wwdt_config_t *config`)  
    *Initializes WWDT configure sturcture.*
- void `WWDT_Init` (`WWDT_Type *base`, const `wwdt_config_t *config`)  
    *Initializes the WWDT.*
- void `WWDT_Deinit` (`WWDT_Type *base`)  
    *Shuts down the WWDT.*

## WWDT Functional Operation

- static void `WWDT_Enable` (`WWDT_Type *base`)  
    *Enables the WWDT module.*
- static void `WWDT_Disable` (`WWDT_Type *base`)  
    *Disables the WWDT module.*
- static uint32\_t `WWDT_GetStatusFlags` (`WWDT_Type *base`)  
    *Gets all WWDT status flags.*
- void `WWDT_ClearStatusFlags` (`WWDT_Type *base`, uint32\_t mask)  
    *Clear WWDT flag.*
- static void `WWDT_SetWarningValue` (`WWDT_Type *base`, uint32\_t warningValue)  
    *Set the WWDT warning value.*
- static void `WWDT_SetTimeoutValue` (`WWDT_Type *base`, uint32\_t timeoutCount)  
    *Set the WWDT timeout value.*
- static void `WWDT_SetWindowValue` (`WWDT_Type *base`, uint32\_t windowValue)  
    *Sets the WWDT window value.*
- void `WWDT_Refresh` (`WWDT_Type *base`)  
    *Refreshes the WWDT timer.*



## 12.4 Data Structure Documentation

### 12.4.1 struct wwdt\_config\_t

#### Data Fields

- bool [enableWwdt](#)  
*Enables or disables WWDT.*
- bool [enableWatchdogReset](#)  
*true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset*
- bool [enableWatchdogProtect](#)  
*true: Enable watchdog protect; timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time*
- bool [enableLockOscillator](#)  
*true: Disabling or powering down the watchdog oscillator is prevented Once set, this bit can only be cleared by a reset false: Do not lock oscillator*
- uint32\_t [windowValue](#)  
*Window value, set this to 0xFFFFFFFF if windowing is not in effect.*
- uint32\_t [timeoutValue](#)  
*Timeout value.*
- uint32\_t [warningValue](#)  
*Watchdog time counter value that will generate a warning interrupt.*

#### 12.4.1.0.0.21 Field Documentation

##### 12.4.1.0.0.21.1 uint32\_t wwdt\_config\_t::warningValue

Set this to 0 for no warning

## 12.5 Macro Definition Documentation

### 12.5.1 #define FSL\_WWDT\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

## 12.6 Enumeration Type Documentation

### 12.6.1 enum \_wwdt\_status\_flags\_t

This structure contains the WWDT status flags for use in the WWDT functions.

Enumerator

*kWWDT\_TimeoutFlag* Time-out flag, set when the timer times out.

*kWWDT\_WarningFlag* Warning interrupt flag, set when timer is below the value WDWARNINT.

## 12.7 Function Documentation

### 12.7.1 void WWDT\_GetDefaultConfig ( wwdt\_config\_t \* config )

This function initializes the WWDT configure structure to default value. The default value are:

## Function Documentation

```
* config->enableWwdt = true;
* config->enableWatchdogReset = false;
* config->enableWatchdogProtect = false;
* config->enableLockOscillator = false;
* config->windowValue = 0xFFFFFU;
* config->timeoutValue = 0xFFFFFU;
* config->warningValue = 0;
*
```

### Parameters

<i>config</i>	Pointer to WWDT config structure.
---------------	-----------------------------------

### See Also

[wwdt\\_config\\_t](#)

### 12.7.2 void WWDT\_Init ( WWDT\_Type \* *base*, const wwdt\_config\_t \* *config* )

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
* wwdt_config_t config;
* WWDT_GetDefaultConfig(&config);
* config.timeoutValue = 0x7ffU;
* WWDT_Init(wwdt_base, &config);
*
```

### Parameters

<i>base</i>	WWDT peripheral base address
<i>config</i>	The configuration of WWDT

### 12.7.3 void WWDT\_Deinit ( WWDT\_Type \* *base* )

This function shuts down the WWDT.

### Parameters

---

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

#### 12.7.4 static void WWDT\_Enable ( WWDT\_Type \* *base* ) [inline], [static]

This function write value into WWDT\_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

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

#### 12.7.5 static void WWDT\_Disable ( WWDT\_Type \* *base* ) [inline], [static]

This function write value into WWDT\_MOD register to disable the WWDT.

Parameters

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

#### 12.7.6 static uint32\_t WWDT\_GetStatusFlags ( WWDT\_Type \* *base* ) [inline], [static]

This function gets all status flags.

Example for getting Timeout Flag:

```
*  uint32_t status;
*  status = WWDT_GetStatusFlags(wwdt_base) &
*           kWWDT_TimeoutFlag;
*
```

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [\\_wwdt\\_status\\_flags\\_t](#)

## Function Documentation

### 12.7.7 void WWDT\_ClearStatusFlags ( WWDT\_Type \* *base*, uint32\_t *mask* )

This function clears WWDT status flag.

Example for clearing warning flag:

```
* WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);  
*
```

#### Parameters

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

### 12.7.8 static void WWDT\_SetWarningValue ( WWDT\_Type \* *base*, uint32\_t *warningValue* ) [inline], [static]

The WDWARNINT register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by WARNINT, an interrupt will be generated after the subsequent WDCLK.

#### Parameters

<i>base</i>	WWDT peripheral base address
<i>warningValue</i>	WWDT warning value.

### 12.7.9 static void WWDT\_SetTimeoutValue ( WWDT\_Type \* *base*, uint32\_t *timeoutCount* ) [inline], [static]

This function sets the timeout value. Every time a feed sequence occurs the value in the TC register is loaded into the Watchdog timer. Writing a value below 0xFF will cause 0xFF to be loaded into the TC register. Thus the minimum time-out interval is TWDCLK\*256\*4. If enableWatchdogProtect flag is true in [wwdt\\_config\\_t](#) config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the WDTOF flag.

#### Parameters

<i>base</i>	WWDT peripheral base address
<i>timeoutCount</i>	WWDT timeout value, count of WWDT clock tick.

#### 12.7.10 static void WWDT\_SetWindowValue ( WWDT\_Type \* *base*, uint32\_t *windowValue* ) [inline], [static]

The WINDOW register determines the highest TV value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in WINDOW, a watchdog event will occur. To disable windowing, set *windowValue* to 0xFFFFFFFF (maximum possible timer value) so windowing is not in effect.

Parameters

<i>base</i>	WWDT peripheral base address
<i>windowValue</i>	WWDT window value.

#### 12.7.11 void WWDT\_Refresh ( WWDT\_Type \* *base* )

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

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



## Chapter 13

### RTC: Real Time Clock

#### 13.1 Overview

The MCUXpresso SDK provides a driver for the Real Time Clock (RTC).

#### 13.2 Function groups

The RTC driver supports operating the module as a time counter.

##### 13.2.1 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.

##### 13.2.2 Set & Get Datetime

The function [RTC\\_SetDatetime\(\)](#) sets the timer period in seconds. User passes 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.

##### 13.2.3 Set & Get Alarm

The function [RTC\\_SetAlarm\(\)](#) sets the alarm time period in seconds. User passes 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.

##### 13.2.4 Start & Stop timer

The function [RTC\\_StartTimer\(\)](#) starts the RTC time counter.

The function [RTC\\_StopTimer\(\)](#) stops the RTC time counter.

## Typical use case

### 13.2.5 Status

Provides functions to get and clear the RTC status.

### 13.2.6 Interrupt

Provides functions to enable/disable RTC interrupts and get current enabled interrupts.

### 13.2.7 High resolution timer

Provides functions to enable high resolution timer and set and get the wake time.

## 13.3 Typical use case

### 13.3.1 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`

## Files

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

## Data Structures

- struct [rtc\\_datetime\\_t](#)  
*Structure is used to hold the date and time. [More...](#)*

## Enumerations

- enum [rtc\\_interrupt\\_enable\\_t](#) {  
    [kRTC\\_AlarmInterruptEnable](#) = RTC\_CTRL\_ALARMDPD\_EN\_MASK,  
    [kRTC\\_WakeupInterruptEnable](#) = RTC\_CTRL\_WAKEDPD\_EN\_MASK }  
*List of RTC interrupts.*
- enum [rtc\\_status\\_flags\\_t](#) {  
    [kRTC\\_AlarmFlag](#) = RTC\_CTRL\_ALARM1HZ\_MASK,  
    [kRTC\\_WakeupFlag](#) = RTC\_CTRL\_WAKE1KHZ\_MASK }  
*List of RTC flags.*

## Functions

- static void [RTC\\_SetWakeupCount](#) (RTC\_Type \*base, uint16\_t wakeupValue)  
*Enable the RTC high resolution timer and set the wake-up time.*
- static uint16\_t [RTC\\_GetWakeupCount](#) (RTC\_Type \*base)  
*Read actual RTC counter value.*



- static void [RTC\\_Reset](#) (RTC\_Type \*base)  
*Performs a software reset on the RTC module.*

## Driver version

- #define [FSL\\_RTC\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 0))  
*Version 2.0.0.*

## Initialization and deinitialization

- void [RTC\\_Init](#) (RTC\_Type \*base)  
*Ungates the RTC clock and enables the RTC oscillator.*
- static void [RTC\\_Deinit](#) (RTC\_Type \*base)  
*Stop the timer and gate the RTC clock.*

## Current Time & Alarm

- [status\\_t RTC\\_SetDatetime](#) (RTC\_Type \*base, const [rtc\\_datetime\\_t](#) \*datetime)  
*Sets the RTC date and time according to the given time structure.*
- void [RTC\\_GetDatetime](#) (RTC\_Type \*base, [rtc\\_datetime\\_t](#) \*datetime)  
*Gets the RTC time and stores it in the given time structure.*
- [status\\_t RTC\\_SetAlarm](#) (RTC\_Type \*base, const [rtc\\_datetime\\_t](#) \*alarmTime)  
*Sets the RTC alarm time.*
- void [RTC\\_GetAlarm](#) (RTC\_Type \*base, [rtc\\_datetime\\_t](#) \*datetime)  
*Returns the RTC alarm time.*

## Interrupt Interface

- static void [RTC\\_EnableInterrupts](#) (RTC\_Type \*base, uint32\_t mask)  
*Enables the selected RTC interrupts.*
- static void [RTC\\_DisableInterrupts](#) (RTC\_Type \*base, uint32\_t mask)  
*Disables the selected RTC interrupts.*
- static uint32\_t [RTC\\_GetEnabledInterrupts](#) (RTC\_Type \*base)  
*Gets the enabled RTC interrupts.*

## Status Interface

- static uint32\_t [RTC\\_GetStatusFlags](#) (RTC\_Type \*base)  
*Gets the RTC status flags.*
- static 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.*

## Enumeration Type Documentation

### 13.4 Data Structure Documentation

#### 13.4.1 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.*

##### 13.4.1.0.0.22 Field Documentation

13.4.1.0.0.22.1 uint16\_t rtc\_datetime\_t::year

13.4.1.0.0.22.2 uint8\_t rtc\_datetime\_t::month

13.4.1.0.0.22.3 uint8\_t rtc\_datetime\_t::day

13.4.1.0.0.22.4 uint8\_t rtc\_datetime\_t::hour

13.4.1.0.0.22.5 uint8\_t rtc\_datetime\_t::minute

13.4.1.0.0.22.6 uint8\_t rtc\_datetime\_t::second

### 13.5 Enumeration Type Documentation

#### 13.5.1 enum rtc\_interrupt\_enable\_t

Enumerator

*kRTC\_AlarmInterruptEnable* Alarm interrupt.

*kRTC\_WakeupInterruptEnable* Wake-up interrupt.

#### 13.5.2 enum rtc\_status\_flags\_t

Enumerator

*kRTC\_AlarmFlag* Alarm flag.

*kRTC\_WakeupFlag* 1kHz wake-up timer flag

## 13.6 Function Documentation

### 13.6.1 void RTC\_Init ( RTC\_Type \* *base* )

Note

This API should be called at the beginning of the application using the RTC driver.

Parameters

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

### 13.6.2 static void RTC\_Deinit ( RTC\_Type \* *base* ) [inline], [static]

Parameters

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

### 13.6.3 status\_t RTC\_SetDatetime ( RTC\_Type \* *base*, const rtc\_datetime\_t \* *datetime* )

The RTC counter must be stopped prior to calling this function as writes to the RTC seconds register will fail if the RTC counter is running.

Parameters

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to structure where the date and time details to set are stored

Returns

kStatus\_Success: Success in setting the time and starting the RTC  
 kStatus\_InvalidArgument: Error because the datetime format is incorrect

### 13.6.4 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 structure where the date and time details are stored.

### 13.6.5 **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 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

### 13.6.6 **void RTC\_GetAlarm ( RTC\_Type \* *base*, rtc\_datetime\_t \* *datetime* )**

### Parameters

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to structure where the alarm date and time details are stored.

### 13.6.7 **static void RTC\_SetWakeupCount ( RTC\_Type \* *base*, uint16\_t *wakeupValue* ) [inline], [static]**

### Parameters

<i>base</i>	RTC peripheral base address
<i>wakeupValue</i>	The value to be loaded into the RTC WAKE register

### 13.6.8 static uint16\_t RTC\_GetWakeupCount ( RTC\_Type \* *base* ) [inline], [static]

Parameters

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

### 13.6.9 static void RTC\_EnableInterrupts ( RTC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

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

### 13.6.10 static void RTC\_DisableInterrupts ( RTC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

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

### 13.6.11 static uint32\_t RTC\_GetEnabledInterrupts ( RTC\_Type \* *base* ) [inline], [static]

## Function Documentation

### Parameters

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

### Returns

The enabled interrupts. This is the logical OR of members of the enumeration [rtc\\_interrupt\\_enable\\_t](#)

### 13.6.12 static uint32\_t RTC\_GetStatusFlags ( RTC\_Type \* *base* ) [inline], [static]

### Parameters

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

### Returns

The status flags. This is the logical OR of members of the enumeration [rtc\\_status\\_flags\\_t](#)

### 13.6.13 static void RTC\_ClearStatusFlags ( RTC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

### 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 <a href="#">rtc_status_flags_t</a>

### 13.6.14 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
-------------	-----------------------------

### 13.6.15 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
-------------	-----------------------------

### 13.6.16 static void RTC\_Reset ( RTC\_Type \* *base* ) [inline], [static]

This resets all RTC registers to their reset value. The bit is cleared by software explicitly clearing it.

Parameters

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





## Chapter 14

### MRT: Multi-Rate Timer

#### 14.1 Overview

The MCUXpresso SDK provides a driver for the Multi-Rate Timer (MRT) of MCUXpresso SDK devices.

#### 14.2 Function groups

The MRT driver supports operating the module as a time counter.

##### 14.2.1 Initialization and deinitialization

The function [MRT\\_Init\(\)](#) initializes the MRT with specified configurations. The function [MRT\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the MRT operating mode.

The function [MRT\\_Deinit\(\)](#) stops the MRT timers and disables the module clock.

##### 14.2.2 Timer period Operations

The function [MRT\\_UpdateTimerPeriod\(\)](#) is used to update the timer period in units of count. The new value will be immediately loaded or will be loaded at the end of the current time interval.

The function [MRT\\_GetCurrentTimerCount\(\)](#) reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. User can call the utility macros provided in `fsl_common.h` to convert to microseconds or milliseconds

##### 14.2.3 Start and Stop timer operations

The function [MRT\\_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer loads the period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function [MRT\\_StopTimer\(\)](#) stops the timer counting.

## Typical use case

### 14.2.4 Get and release channel

These functions can be used to reserve and release a channel. The function [MRT\\_GetIdleChannel\(\)](#) finds the available channel. This function returns the lowest available channel number. The function [MRT\\_ReleaseChannel\(\)](#) release the channel when the timer is using the multi-task mode. In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use.

### 14.2.5 Status

Provides functions to get and clear the PIT status.

### 14.2.6 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

## 14.3 Typical use case

### 14.3.1 MRT tick example

Updates the MRT period and toggles an LED periodically. Refer to the driver examples codes located at `<SDK_ROOT>/boards/<BOARD>/driver_examples/mrt`

## Files

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

## Data Structures

- struct [mrt\\_config\\_t](#)  
*MRT configuration structure. [More...](#)*

## Enumerations

- enum [mrt\\_chnl\\_t](#) {  
    [kMRT\\_Channel\\_0](#) = 0U,  
    [kMRT\\_Channel\\_1](#),  
    [kMRT\\_Channel\\_2](#),  
    [kMRT\\_Channel\\_3](#) }  
*List of MRT channels.*
- enum [mrt\\_timer\\_mode\\_t](#) {  
    [kMRT\\_RepeatMode](#) = (0 << MRT\_CHANNEL\_CTRL\_MODE\_SHIFT),  
    [kMRT\\_OneShotMode](#) = (1 << MRT\_CHANNEL\_CTRL\_MODE\_SHIFT),  
    [kMRT\\_OneShotStallMode](#) = (2 << MRT\_CHANNEL\_CTRL\_MODE\_SHIFT) }  
*List of MRT timer modes.*

- enum `mrt_interrupt_enable_t` { `kMRT_TimerInterruptEnable` = `MRT_CHANNEL_CTRL_INTEN_MASK` }  
*List of MRT interrupts.*
- enum `mrt_status_flags_t` {  
    `kMRT_TimerInterruptFlag` = `MRT_CHANNEL_STAT_INTFLAG_MASK`,  
    `kMRT_TimerRunFlag` = `MRT_CHANNEL_STAT_RUN_MASK` }  
*List of MRT status flags.*

## Driver version

- #define `FSL_MRT_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*Version 2.0.0.*

## Initialization and deinitialization

- void `MRT_Init` (`MRT_Type *base`, const `mrt_config_t *config`)  
*Ungates the MRT clock and configures the peripheral for basic operation.*
- void `MRT_Deinit` (`MRT_Type *base`)  
*Gate the MRT clock.*
- static void `MRT_GetDefaultConfig` (`mrt_config_t *config`)  
*Fill in the MRT config struct with the default settings.*
- static void `MRT_SetupChannelMode` (`MRT_Type *base`, `mrt_chnl_t channel`, const `mrt_timer_mode_t mode`)  
*Sets up an MRT channel mode.*

## Interrupt Interface

- static void `MRT_EnableInterrupts` (`MRT_Type *base`, `mrt_chnl_t channel`, `uint32_t mask`)  
*Enables the MRT interrupt.*
- static void `MRT_DisableInterrupts` (`MRT_Type *base`, `mrt_chnl_t channel`, `uint32_t mask`)  
*Disables the selected MRT interrupt.*
- static `uint32_t` `MRT_GetEnabledInterrupts` (`MRT_Type *base`, `mrt_chnl_t channel`)  
*Gets the enabled MRT interrupts.*

## Status Interface

- static `uint32_t` `MRT_GetStatusFlags` (`MRT_Type *base`, `mrt_chnl_t channel`)  
*Gets the MRT status flags.*
- static void `MRT_ClearStatusFlags` (`MRT_Type *base`, `mrt_chnl_t channel`, `uint32_t mask`)  
*Clears the MRT status flags.*

## Read and Write the timer period

- void `MRT_UpdateTimerPeriod` (`MRT_Type *base`, `mrt_chnl_t channel`, `uint32_t count`, bool `immediateLoad`)  
*Used to update the timer period in units of count.*
- static `uint32_t` `MRT_GetCurrentTimerCount` (`MRT_Type *base`, `mrt_chnl_t channel`)  
*Reads the current timer counting value.*

## Enumeration Type Documentation

### Timer Start and Stop

- static void [MRT\\_StartTimer](#) (MRT\_Type \*base, [mrt\\_chnl\\_t](#) channel, uint32\_t count)  
*Starts the timer counting.*
- static void [MRT\\_StopTimer](#) (MRT\_Type \*base, [mrt\\_chnl\\_t](#) channel)  
*Stops the timer counting.*

### Get & release channel

- static uint32\_t [MRT\\_GetIdleChannel](#) (MRT\_Type \*base)  
*Find the available channel.*
- static void [MRT\\_ReleaseChannel](#) (MRT\_Type \*base, [mrt\\_chnl\\_t](#) channel)  
*Release the channel when the timer is using the multi-task mode.*

## 14.4 Data Structure Documentation

### 14.4.1 struct mrt\_config\_t

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the [MRT\\_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 [enableMultiTask](#)  
*true: Timers run in multi-task mode; false: Timers run in hardware status mode*

## 14.5 Enumeration Type Documentation

### 14.5.1 enum mrt\_chnl\_t

Enumerator

***kMRT\_Channel\_0*** MRT channel number 0.  
***kMRT\_Channel\_1*** MRT channel number 1.  
***kMRT\_Channel\_2*** MRT channel number 2.  
***kMRT\_Channel\_3*** MRT channel number 3.

### 14.5.2 enum mrt\_timer\_mode\_t

Enumerator

***kMRT\_RepeatMode*** Repeat Interrupt mode.  
***kMRT\_OneShotMode*** One-shot Interrupt mode.  
***kMRT\_OneShotStallMode*** One-shot stall mode.

### 14.5.3 enum mrt\_interrupt\_enable\_t

Enumerator

*kMRT\_TimerInterruptEnable* Timer interrupt enable.

### 14.5.4 enum mrt\_status\_flags\_t

Enumerator

*kMRT\_TimerInterruptFlag* Timer interrupt flag.

*kMRT\_TimerRunFlag* Indicates state of the timer.

## 14.6 Function Documentation

### 14.6.1 void MRT\_Init ( MRT\_Type \* *base*, const mrt\_config\_t \* *config* )

Note

This API should be called at the beginning of the application using the MRT driver.

Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>config</i>	Pointer to user's MRT config structure

### 14.6.2 void MRT\_Deinit ( MRT\_Type \* *base* )

Parameters

<i>base</i>	Multi-Rate timer peripheral base address
-------------	--

### 14.6.3 static void MRT\_GetDefaultConfig ( mrt\_config\_t \* *config* ) [inline], [static]

The default values are:

```
* config->enableMultiTask = false;
*
```

## Function Documentation

### Parameters

<i>config</i>	Pointer to user's MRT config structure.
---------------	---

#### 14.6.4 static void MRT\_SetupChannelMode ( MRT\_Type \* *base*, mrt\_chnl\_t *channel*, const mrt\_timer\_mode\_t *mode* ) [inline], [static]

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Channel that is being configured.
<i>mode</i>	Timer mode to use for the channel.

#### 14.6.5 static void MRT\_EnableInterrupts ( MRT\_Type \* *base*, mrt\_chnl\_t *channel*, uint32\_t *mask* ) [inline], [static]

### Parameters

<i>base</i>	Multi-Rate timer 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="#">mrt_interrupt_enable_t</a>

#### 14.6.6 static void MRT\_DisableInterrupts ( MRT\_Type \* *base*, mrt\_chnl\_t *channel*, uint32\_t *mask* ) [inline], [static]

### Parameters

<i>base</i>	Multi-Rate timer 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="#">mrt_interrupt_enable_t</a>

**14.6.7** `static uint32_t MRT_GetEnabledInterrupts ( MRT_Type * base, mrt_chnl_t channel ) [inline], [static]`

## Function Documentation

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number

### Returns

The enabled interrupts. This is the logical OR of members of the enumeration [mrt\\_interrupt\\_enable\\_t](#)

#### 14.6.8 static uint32\_t MRT\_GetStatusFlags ( MRT\_Type \* *base*, mrt\_chnl\_t *channel* ) [inline], [static]

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number

### Returns

The status flags. This is the logical OR of members of the enumeration [mrt\\_status\\_flags\\_t](#)

#### 14.6.9 static void MRT\_ClearStatusFlags ( MRT\_Type \* *base*, mrt\_chnl\_t *channel*, uint32\_t *mask* ) [inline], [static]

### Parameters

<i>base</i>	Multi-Rate timer 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="#">mrt_status_flags_t</a>

#### 14.6.10 void MRT\_UpdateTimerPeriod ( MRT\_Type \* *base*, mrt\_chnl\_t *channel*, uint32\_t *count*, bool *immediateLoad* )

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.



## Note

User can call the utility macros provided in fsl\_common.h to convert to ticks

## Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number
<i>count</i>	Timer period in units of ticks
<i>immediateLoad</i>	true: Load the new value immediately into the TIMER register; false: Load the new value at the end of current timer interval

#### 14.6.11 static uint32\_t MRT\_GetCurrentTimerCount ( MRT\_Type \* *base*, mrt\_chnl\_t *channel* ) [inline], [static]

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

## Note

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

## Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number

## Returns

Current timer counting value in ticks

#### 14.6.12 static void MRT\_StartTimer ( MRT\_Type \* *base*, mrt\_chnl\_t *channel*, uint32\_t *count* ) [inline], [static]

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

## Note

User can call the utility macros provided in fsl\_common.h to convert to ticks

## Function Documentation

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number.
<i>count</i>	Timer period in units of ticks

#### 14.6.13 static void MRT\_StopTimer ( MRT\_Type \* *base*, mrt\_chnl\_t *channel* ) [inline], [static]

This function stops the timer from counting.

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number.

#### 14.6.14 static uint32\_t MRT\_GetIdleChannel ( MRT\_Type \* *base* ) [inline], [static]

This function returns the lowest available channel number.

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
-------------	--

#### 14.6.15 static void MRT\_ReleaseChannel ( MRT\_Type \* *base*, mrt\_chnl\_t *channel* ) [inline], [static]

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling [MRT\\_GetIdleChannel\(\)](#) for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

### Parameters

<i>base</i>	Multi-Rate timer peripheral base address
<i>channel</i>	Timer channel number.



## Chapter 15

# ADC: 12-bit SAR Analog-to-Digital Converter Driver

### 15.1 Overview

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

### 15.2 Typical use case

#### 15.2.1 Polling Configuration

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

#### 15.2.2 Interrupt Configuration

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

### Files

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

### Data Structures

- struct [adc\\_config\\_t](#)  
*Define structure for configuring the block. [More...](#)*
- struct [adc\\_conv\\_seq\\_config\\_t](#)  
*Define structure for configuring conversion sequence. [More...](#)*
- struct [adc\\_result\\_info\\_t](#)  
*Define structure of keeping conversion result information. [More...](#)*

### Enumerations

- enum `_adc_status_flags` {
  - `kADC_ThresholdCompareFlagOnChn0` = 1U << 0U,
  - `kADC_ThresholdCompareFlagOnChn1` = 1U << 1U,
  - `kADC_ThresholdCompareFlagOnChn2` = 1U << 2U,
  - `kADC_ThresholdCompareFlagOnChn3` = 1U << 3U,
  - `kADC_ThresholdCompareFlagOnChn4` = 1U << 4U,
  - `kADC_ThresholdCompareFlagOnChn5` = 1U << 5U,
  - `kADC_ThresholdCompareFlagOnChn6` = 1U << 6U,
  - `kADC_ThresholdCompareFlagOnChn7` = 1U << 7U,
  - `kADC_ThresholdCompareFlagOnChn8` = 1U << 8U,
  - `kADC_ThresholdCompareFlagOnChn9` = 1U << 9U,
  - `kADC_ThresholdCompareFlagOnChn10` = 1U << 10U,
  - `kADC_ThresholdCompareFlagOnChn11` = 1U << 11U,
  - `kADC_OverrunFlagForChn0`,
  - `kADC_OverrunFlagForChn1`,
  - `kADC_OverrunFlagForChn2`,
  - `kADC_OverrunFlagForChn3`,
  - `kADC_OverrunFlagForChn4`,
  - `kADC_OverrunFlagForChn5`,
  - `kADC_OverrunFlagForChn6`,
  - `kADC_OverrunFlagForChn7`,
  - `kADC_OverrunFlagForChn8`,
  - `kADC_OverrunFlagForChn9`,
  - `kADC_OverrunFlagForChn10`,
  - `kADC_OverrunFlagForChn11`,
  - `kADC_GlobalOverrunFlagForSeqA` = 1U << 24U,
  - `kADC_GlobalOverrunFlagForSeqB` = 1U << 25U,
  - `kADC_ConvSeqAInterruptFlag` = 1U << 28U,
  - `kADC_ConvSeqBInterruptFlag` = 1U << 29U,
  - `kADC_ThresholdCompareInterruptFlag` = 1U << 30U,
  - `kADC_OverrunInterruptFlag` = 1U << 31U }

*Flags.*

- enum `_adc_interrupt_enable` {
  - `kADC_ConvSeqAInterruptEnable` = ADC\_INTEN\_SEQA\_INTEN\_MASK,
  - `kADC_ConvSeqBInterruptEnable` = ADC\_INTEN\_SEQB\_INTEN\_MASK,
  - `kADC_OverrunInterruptEnable` = ADC\_INTEN\_OVR\_INTEN\_MASK }

*Interrupts.*

- enum `adc_trigger_polarity_t` {
  - `kADC_TriggerPolarityNegativeEdge` = 0U,
  - `kADC_TriggerPolarityPositiveEdge` = 1U }

*Define selection of polarity of selected input trigger for conversion sequence.*

- enum `adc_priority_t` {
  - `kADC_PriorityLow` = 0U,
  - `kADC_PriorityHigh` = 1U }

- *Define selection of conversion sequence's priority.*  
enum `adc_seq_interrupt_mode_t` {  
    `kADC_InterruptForEachConversion` = 0U,  
    `kADC_InterruptForEachSequence` = 1U }
- *Define selection of conversion sequence's interrupt.*  
enum `adc_threshold_compare_status_t` {  
    `kADC_ThresholdCompareInRange` = 0U,  
    `kADC_ThresholdCompareBelowRange` = 1U,  
    `kADC_ThresholdCompareAboveRange` = 2U }
- *Define status of threshold compare result.*  
enum `adc_threshold_crossing_status_t` {  
    `kADC_ThresholdCrossingNoDetected` = 0U,  
    `kADC_ThresholdCrossingDownward` = 2U,  
    `kADC_ThresholdCrossingUpward` = 3U }
- *Define status of threshold crossing detection result.*  
enum `adc_threshold_interrupt_mode_t` {  
    `kADC_ThresholdInterruptDisabled` = 0U,  
    `kADC_ThresholdInterruptOnOutside` = 1U,  
    `kADC_ThresholdInterruptOnCrossing` = 2U }
- *Define interrupt mode for threshold compare event.*

## Driver version

- #define `LPC_ADC_DRIVER_VERSION` (`MAKE_VERSION(2, 2, 0)`)  
ADC driver version 2.2.0.

## Initialization and Deinitialization

- void `ADC_Init` (`ADC_Type *base`, const `adc_config_t *config`)  
*Initialize the ADC module.*
- void `ADC_Deinit` (`ADC_Type *base`)  
*Deinitialize the ADC module.*
- void `ADC_GetDefaultConfig` (`adc_config_t *config`)  
*Gets an available pre-defined settings for initial configuration.*
- bool `ADC_DoSelfCalibration` (`ADC_Type *base`, `uint32_t frequency`)  
*Do the self calibration.*
- static void `ADC_EnableTemperatureSensor` (`ADC_Type *base`, bool enable)  
*Enable the internal temperature sensor measurement.*

## Control conversion sequence A.

- static void `ADC_EnableConvSeqA` (`ADC_Type *base`, bool enable)  
*Enable the conversion sequence A.*
- void `ADC_SetConvSeqAConfig` (`ADC_Type *base`, const `adc_conv_seq_config_t *config`)  
*Configure the conversion sequence A.*
- static void `ADC_DoSoftwareTriggerConvSeqA` (`ADC_Type *base`)  
*Do trigger the sequence's conversion by software.*
- static void `ADC_EnableConvSeqABurstMode` (`ADC_Type *base`, bool enable)  
*Enable the burst conversion of sequence A.*

## Typical use case

- static void [ADC\\_SetConvSeqAHighPriority](#) (ADC\_Type \*base)  
*Set the high priority for conversion sequence A.*

## Control conversion sequence B.

- static void [ADC\\_EnableConvSeqB](#) (ADC\_Type \*base, bool enable)  
*Enable the conversion sequence B.*
- void [ADC\\_SetConvSeqBConfig](#) (ADC\_Type \*base, const [adc\\_conv\\_seq\\_config\\_t](#) \*config)  
*Configure the conversion sequence B.*
- static void [ADC\\_DoSoftwareTriggerConvSeqB](#) (ADC\_Type \*base)  
*Do trigger the sequence's conversion by software.*
- static void [ADC\\_EnableConvSeqBBurstMode](#) (ADC\_Type \*base, bool enable)  
*Enable the burst conversion of sequence B.*
- static void [ADC\\_SetConvSeqBHighPriority](#) (ADC\_Type \*base)  
*Set the high priority for conversion sequence B.*

## Data result.

- bool [ADC\\_GetConvSeqAGlobalConversionResult](#) (ADC\_Type \*base, [adc\\_result\\_info\\_t](#) \*info)  
*Get the global ADC conversion information of sequence A.*
- bool [ADC\\_GetConvSeqBGlobalConversionResult](#) (ADC\_Type \*base, [adc\\_result\\_info\\_t](#) \*info)  
*Get the global ADC conversion information of sequence B.*
- bool [ADC\\_GetChannelConversionResult](#) (ADC\_Type \*base, uint32\_t channel, [adc\\_result\\_info\\_t](#) \*info)  
*Get the channel's ADC conversion completed under each conversion sequence.*

## Threshold function.

- static void [ADC\\_SetThresholdPair0](#) (ADC\_Type \*base, uint32\_t lowValue, uint32\_t highValue)  
*Set the threshold pair 0 with low and high value.*
- static void [ADC\\_SetThresholdPair1](#) (ADC\_Type \*base, uint32\_t lowValue, uint32\_t highValue)  
*Set the threshold pair 1 with low and high value.*
- static void [ADC\\_SetChannelWithThresholdPair0](#) (ADC\_Type \*base, uint32\_t channelMask)  
*Set given channels to apply the threshold pair 0.*
- static void [ADC\\_SetChannelWithThresholdPair1](#) (ADC\_Type \*base, uint32\_t channelMask)  
*Set given channels to apply the threshold pair 1.*

## Interrupts.

- static void [ADC\\_EnableInterrupts](#) (ADC\_Type \*base, uint32\_t mask)  
*Enable interrupts for conversion sequences.*
- static void [ADC\\_DisableInterrupts](#) (ADC\_Type \*base, uint32\_t mask)  
*Disable interrupts for conversion sequence.*
- static void [ADC\\_EnableShresholdCompareInterrupt](#) (ADC\_Type \*base, uint32\_t channel, [adc\\_threshold\\_interrupt\\_mode\\_t](#) mode)  
*Enable the interrupt of threshold compare event for each channel.*
- static void [ADC\\_EnableThresholdCompareInterrupt](#) (ADC\_Type \*base, uint32\_t channel, [adc\\_threshold\\_interrupt\\_mode\\_t](#) mode)  
*Enable the interrupt of threshold compare event for each channel.*



**Status.**

- static uint32\_t [ADC\\_GetStatusFlags](#) (ADC\_Type \*base)  
*Get status flags of ADC module.*
- static void [ADC\\_ClearStatusFlags](#) (ADC\_Type \*base, uint32\_t mask)  
*Clear status flags of ADC module.*

**15.3 Data Structure Documentation****15.3.1 struct adc\_config\_t****Data Fields**

- uint32\_t [clockDividerNumber](#)  
*This field is only available when using kADC\_ClockSynchronousMode for "clockMode" field.*

**15.3.1.0.0.23 Field Documentation****15.3.1.0.0.23.1 uint32\_t adc\_config\_t::clockDividerNumber**

The divider would be plused by 1 based on the value in this field. The available range is in 8 bits.

**15.3.2 struct adc\_conv\_seq\_config\_t****Data Fields**

- uint32\_t [channelMask](#)  
*Selects which one or more of the ADC channels will be sampled and converted when this sequence is launched.*
- uint32\_t [triggerMask](#)  
*Selects which one or more of the available hardware trigger sources will cause this conversion sequence to be initiated.*
- [adc\\_trigger\\_polarity\\_t](#) [triggerPolarity](#)  
*Select the trigger to launch conversion sequence.*
- bool [enableSyncBypass](#)  
*To enable this feature allows the hardware trigger input to bypass synchronization flip-flop stages and therefore shorten the time between the trigger input signal and the start of a conversion.*
- bool [enableSingleStep](#)  
*When enabling this feature, a trigger will launch a single conversion on the next channel in the sequence instead of the default response of launching an entire sequence of conversions.*
- [adc\\_seq\\_interrupt\\_mode\\_t](#) [interruptMode](#)  
*Select the interrupt/DMA trigger mode.*

## Macro Definition Documentation

### 15.3.2.0.0.24 Field Documentation

#### 15.3.2.0.0.24.1 uint32\_t adc\_conv\_seq\_config\_t::channelMask

The masked channels would be involved in current conversion sequence, beginning with the lowest-order. The available range is in 12-bit.

#### 15.3.2.0.0.24.2 uint32\_t adc\_conv\_seq\_config\_t::triggerMask

The available range is 6-bit.

#### 15.3.2.0.0.24.3 adc\_trigger\_polarity\_t adc\_conv\_seq\_config\_t::triggerPolarity

#### 15.3.2.0.0.24.4 bool adc\_conv\_seq\_config\_t::enableSyncBypass

#### 15.3.2.0.0.24.5 bool adc\_conv\_seq\_config\_t::enableSingleStep

#### 15.3.2.0.0.24.6 adc\_seq\_interrupt\_mode\_t adc\_conv\_seq\_config\_t::interruptMode

### 15.3.3 struct adc\_result\_info\_t

#### Data Fields

- uint32\_t [result](#)  
*Keep the conversion data value.*
- [adc\\_threshold\\_compare\\_status\\_t](#) [thresholdCompareStatus](#)  
*Keep the threshold compare status.*
- [adc\\_threshold\\_crossing\\_status\\_t](#) [thresholdCorssingStatus](#)  
*Keep the threshold crossing status.*
- uint32\_t [channelNumber](#)  
*Keep the channel number for this conversion.*
- bool [overrunFlag](#)  
*Keep the status whether the conversion is overrun or not.*

### 15.3.3.0.0.25 Field Documentation

#### 15.3.3.0.0.25.1 uint32\_t adc\_result\_info\_t::result

#### 15.3.3.0.0.25.2 adc\_threshold\_compare\_status\_t adc\_result\_info\_t::thresholdCompareStatus

#### 15.3.3.0.0.25.3 adc\_threshold\_crossing\_status\_t adc\_result\_info\_t::thresholdCorssingStatus

#### 15.3.3.0.0.25.4 uint32\_t adc\_result\_info\_t::channelNumber

#### 15.3.3.0.0.25.5 bool adc\_result\_info\_t::overrunFlag

## 15.4 Macro Definition Documentation

### 15.4.1 #define LPC\_ADC\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 0))

## 15.5 Enumeration Type Documentation

### 15.5.1 enum \_adc\_status\_flags

Enumerator

<b><i>kADC_ThresholdCompareFlagOnChn0</i></b>	Threshold comparison event on Channel 0.
<b><i>kADC_ThresholdCompareFlagOnChn1</i></b>	Threshold comparison event on Channel 1.
<b><i>kADC_ThresholdCompareFlagOnChn2</i></b>	Threshold comparison event on Channel 2.
<b><i>kADC_ThresholdCompareFlagOnChn3</i></b>	Threshold comparison event on Channel 3.
<b><i>kADC_ThresholdCompareFlagOnChn4</i></b>	Threshold comparison event on Channel 4.
<b><i>kADC_ThresholdCompareFlagOnChn5</i></b>	Threshold comparison event on Channel 5.
<b><i>kADC_ThresholdCompareFlagOnChn6</i></b>	Threshold comparison event on Channel 6.
<b><i>kADC_ThresholdCompareFlagOnChn7</i></b>	Threshold comparison event on Channel 7.
<b><i>kADC_ThresholdCompareFlagOnChn8</i></b>	Threshold comparison event on Channel 8.
<b><i>kADC_ThresholdCompareFlagOnChn9</i></b>	Threshold comparison event on Channel 9.
<b><i>kADC_ThresholdCompareFlagOnChn10</i></b>	Threshold comparison event on Channel 10.
<b><i>kADC_ThresholdCompareFlagOnChn11</i></b>	Threshold comparison event on Channel 11.
<b><i>kADC_OverrunFlagForChn0</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 0.
<b><i>kADC_OverrunFlagForChn1</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 1.
<b><i>kADC_OverrunFlagForChn2</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 2.
<b><i>kADC_OverrunFlagForChn3</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 3.
<b><i>kADC_OverrunFlagForChn4</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 4.
<b><i>kADC_OverrunFlagForChn5</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 5.
<b><i>kADC_OverrunFlagForChn6</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 6.
<b><i>kADC_OverrunFlagForChn7</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 7.
<b><i>kADC_OverrunFlagForChn8</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 8.
<b><i>kADC_OverrunFlagForChn9</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 9.
<b><i>kADC_OverrunFlagForChn10</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 10.
<b><i>kADC_OverrunFlagForChn11</i></b>	Mirror the OVERRUN status flag from the result register for ADC channel 11.
<b><i>kADC_GlobalOverrunFlagForSeqA</i></b>	Mirror the glabal OVERRUN status flag for conversion sequence A.
<b><i>kADC_GlobalOverrunFlagForSeqB</i></b>	Mirror the global OVERRUN status flag for conversion sequence B.

## Enumeration Type Documentation

*kADC\_ConvSeqAInterruptFlag* Sequence A interrupt/DMA trigger.  
*kADC\_ConvSeqBInterruptFlag* Sequence B interrupt/DMA trigger.  
*kADC\_ThresholdCompareInterruptFlag* Threshold comparison interrupt flag.  
*kADC\_OverrunInterruptFlag* Overrun interrupt flag.

### 15.5.2 enum \_adc\_interrupt\_enable

Note

Not all the interrupt options are listed here

Enumerator

*kADC\_ConvSeqAInterruptEnable* Enable interrupt upon completion of each individual conversion in sequence A, or entire sequence.  
*kADC\_ConvSeqBInterruptEnable* Enable interrupt upon completion of each individual conversion in sequence B, or entire sequence.  
*kADC\_OverrunInterruptEnable* Enable the detection of an overrun condition on any of the channel data registers will cause an overrun interrupt/DMA trigger.

### 15.5.3 enum adc\_trigger\_polarity\_t

Enumerator

*kADC\_TriggerPolarityNegativeEdge* A negative edge launches the conversion sequence on the trigger(s).  
*kADC\_TriggerPolarityPositiveEdge* A positive edge launches the conversion sequence on the trigger(s).

### 15.5.4 enum adc\_priority\_t

Enumerator

*kADC\_PriorityLow* This sequence would be preempted when another sequence is started.  
*kADC\_PriorityHigh* This sequence would preempt other sequence even when it is started.

### 15.5.5 enum adc\_seq\_interrupt\_mode\_t

Enumerator

*kADC\_InterruptForEachConversion* The sequence interrupt/DMA trigger will be set at the end of each individual ADC conversion inside this conversion sequence.

***kADC\_InterruptForEachSequence*** The sequence interrupt/DMA trigger will be set when the entire set of this sequence conversions completes.

### 15.5.6 enum adc\_threshold\_compare\_status\_t

Enumerator

***kADC\_ThresholdCompareInRange*** LOW threshold  $\leq$  conversion value  $\leq$  HIGH threshold.

***kADC\_ThresholdCompareBelowRange*** conversion value  $<$  LOW threshold.

***kADC\_ThresholdCompareAboveRange*** conversion value  $>$  HIGH threshold.

### 15.5.7 enum adc\_threshold\_crossing\_status\_t

Enumerator

***kADC\_ThresholdCrossingNoDetected*** No threshold Crossing detected.

***kADC\_ThresholdCrossingDownward*** Downward Threshold Crossing detected.

***kADC\_ThresholdCrossingUpward*** Upward Threshold Crossing Detected.

### 15.5.8 enum adc\_threshold\_interrupt\_mode\_t

Enumerator

***kADC\_ThresholdInterruptDisabled*** Threshold comparison interrupt is disabled.

***kADC\_ThresholdInterruptOnOutside*** Threshold comparison interrupt is enabled on outside threshold.

***kADC\_ThresholdInterruptOnCrossing*** Threshold comparison interrupt is enabled on crossing threshold.

## 15.6 Function Documentation

### 15.6.1 void ADC\_Init ( ADC\_Type \* *base*, const adc\_config\_t \* *config* )

Parameters

---

## Function Documentation

<i>base</i>	ADC peripheral base address.
<i>config</i>	Pointer to configuration structure, see to <a href="#">adc_config_t</a> .

### 15.6.2 void ADC\_Deinit ( ADC\_Type \* *base* )

Parameters

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

### 15.6.3 void ADC\_GetDefaultConfig ( adc\_config\_t \* *config* )

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

```
* config->clockMode = kADC_ClockSynchronousMode;
* config->clockDividerNumber = 0U;
* config->resolution = kADC_Resolution12bit;
* config->enableBypassCalibration = false;
* config->sampleTimeNumber = 0U;
*
```

Parameters

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

### 15.6.4 bool ADC\_DoSelfCalibration ( ADC\_Type \* *base*, uint32\_t *frequency* )

To calibrate the ADC, set the ADC clock to 500 kHz. In order to achieve the specified ADC accuracy, the A/D converter must be recalibrated, at a minimum, following every chip reset before initiating normal ADC operation.

Parameters

<i>base</i>	ADC peripheral base address.
<i>frequency</i>	The system clock frequency to ADC.

Return values

<i>true</i>	Calibration succeed.
<i>false</i>	Calibration failed.

### 15.6.5 static void ADC\_EnableTemperatureSensor ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

When enabling the internal temperature sensor measurement, the channel 0 would be connected to internal sensor instead of external pin.

Parameters

<i>base</i>	ADC peripheral base address.
<i>enable</i>	Switcher to enable the feature or not.

### 15.6.6 static void ADC\_EnableConvSeqA ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

Parameters

<i>base</i>	ADC peripheral base address.
<i>enable</i>	Switcher to enable the feature or not.

### 15.6.7 void ADC\_SetConvSeqAConfig ( ADC\_Type \* *base*, const adc\_conv\_seq\_config\_t \* *config* )

Parameters

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

## Function Documentation

<i>config</i>	Pointer to configuration structure, see to <a href="#">adc_conv_seq_config_t</a> .
---------------	--

### 15.6.8 static void ADC\_DoSoftwareTriggerConvSeqA ( ADC\_Type \* *base* ) [inline], [static]

Parameters

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

### 15.6.9 static void ADC\_EnableConvSeqABurstMode ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before conversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

Parameters

<i>base</i>	ADC peripheral base address.
<i>enable</i>	Switcher to enable this feature.

### 15.6.10 static void ADC\_SetConvSeqAHighPriority ( ADC\_Type \* *base* ) [inline], [static]

Parameters

<i>base</i>	ADC peripheral bass address.
-------------	------------------------------

### 15.6.11 static void ADC\_EnableConvSeqB ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.



## Parameters

<i>base</i>	ADC peripheral base address.
<i>enable</i>	Switcher to enable the feature or not.

### 15.6.12 void ADC\_SetConvSeqBConfig ( ADC\_Type \* *base*, const adc\_conv\_seq\_config\_t \* *config* )

## Parameters

<i>base</i>	ADC peripheral base address.
<i>config</i>	Pointer to configuration structure, see to <a href="#">adc_conv_seq_config_t</a> .

### 15.6.13 static void ADC\_DoSoftwareTriggerConvSeqB ( ADC\_Type \* *base* ) [inline], [static]

## Parameters

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

### 15.6.14 static void ADC\_EnableConvSeqBBurstMode ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before conversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

## Parameters

<i>base</i>	ADC peripheral base address.
<i>enable</i>	Switcher to enable this feature.

### 15.6.15 static void ADC\_SetConvSeqBHighPriority ( ADC\_Type \* *base* ) [inline], [static]

## Function Documentation

### Parameters

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

### 15.6.16 **bool** ADC\_GetConvSeqAGlobalConversionResult ( ADC\_Type \* *base*, adc\_result\_info\_t \* *info* )

### Parameters

<i>base</i>	ADC peripheral base address.
<i>info</i>	Pointer to information structure, see to <a href="#">adc_result_info_t</a> ;

### Return values

<i>true</i>	The conversion result is ready.
<i>false</i>	The conversion result is not ready yet.

### 15.6.17 **bool** ADC\_GetConvSeqBGlobalConversionResult ( ADC\_Type \* *base*, adc\_result\_info\_t \* *info* )

### Parameters

<i>base</i>	ADC peripheral base address.
<i>info</i>	Pointer to information structure, see to <a href="#">adc_result_info_t</a> ;

### Return values

<i>true</i>	The conversion result is ready.
<i>false</i>	The conversion result is not ready yet.

### 15.6.18 **bool** ADC\_GetChannelConversionResult ( ADC\_Type \* *base*, uint32\_t *channel*, adc\_result\_info\_t \* *info* )

## Parameters

<i>base</i>	ADC peripheral base address.
<i>channel</i>	The indicated channel number.
<i>info</i>	Pointer to information structure, see to <a href="#">adc_result_info_t</a> ;

## Return values

<i>true</i>	The conversion result is ready.
<i>false</i>	The conversion result is not ready yet.

**15.6.19 static void ADC\_SetThresholdPair0 ( ADC\_Type \* *base*, uint32\_t *lowValue*, uint32\_t *highValue* ) [inline], [static]**

## Parameters

<i>base</i>	ADC peripheral base address.
<i>lowValue</i>	LOW threshold value.
<i>highValue</i>	HIGH threshold value.

**15.6.20 static void ADC\_SetThresholdPair1 ( ADC\_Type \* *base*, uint32\_t *lowValue*, uint32\_t *highValue* ) [inline], [static]**

## Parameters

<i>base</i>	ADC peripheral base address.
<i>lowValue</i>	LOW threshold value. The available value is with 12-bit.
<i>highValue</i>	HIGH threshold value. The available value is with 12-bit.

**15.6.21 static void ADC\_SetChannelWithThresholdPair0 ( ADC\_Type \* *base*, uint32\_t *channelMask* ) [inline], [static]**

## Function Documentation

### Parameters

<i>base</i>	ADC peripheral base address.
<i>channelMask</i>	Indicated channels' mask.

**15.6.22** `static void ADC_SetChannelWithThresholdPair1 ( ADC_Type * base,  
uint32_t channelMask ) [inline], [static]`

### Parameters

<i>base</i>	ADC peripheral base address.
<i>channelMask</i>	Indicated channels' mask.

**15.6.23** `static void ADC_EnableInterrupts ( ADC_Type * base, uint32_t mask )  
[inline], [static]`

### Parameters

<i>base</i>	ADC peripheral base address.
<i>mask</i>	Mask of interrupt mask value for global block except each channel, see to <a href="#">_adc_interrupt_enable</a> .

**15.6.24** `static void ADC_DisableInterrupts ( ADC_Type * base, uint32_t mask )  
[inline], [static]`

### Parameters

<i>base</i>	ADC peripheral base address.
<i>mask</i>	Mask of interrupt mask value for global block except each channel, see to <a href="#">_adc_interrupt_enable</a> .

**15.6.25**    **static void ADC\_EnableShresholdCompareInterrupt ( ADC\_Type \* *base*,  
uint32\_t *channel*, adc\_threshold\_interrupt\_mode\_t *mode* ) [inline],  
[static]**

**15.6.26**    **static void ADC\_EnableThresholdCompareInterrupt ( ADC\_Type \* *base*,  
uint32\_t *channel*, adc\_threshold\_interrupt\_mode\_t *mode* ) [inline],  
[static]**

## Function Documentation

### Parameters

<i>base</i>	ADC peripheral base address.
<i>channel</i>	Channel number.
<i>mode</i>	Interrupt mode for threshold compare event, see to <a href="#">adc_threshold_interrupt_mode_t</a> .

**15.6.27** `static uint32_t ADC_GetStatusFlags ( ADC_Type * base ) [inline], [static]`

### Parameters

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

### Returns

Mask of status flags of module, see to [\\_adc\\_status\\_flags](#).

**15.6.28** `static void ADC_ClearStatusFlags ( ADC_Type * base, uint32_t mask ) [inline], [static]`

### Parameters

<i>base</i>	ADC peripheral base address.
<i>mask</i>	Mask of status flags of module, see to <a href="#">_adc_status_flags</a> .

## Chapter 16

# FLASHIAP: Flash In Application Programming Driver

### 16.1 Overview

The MCUXpresso SDK provides a driver for the Flash In Application Programming (FLASHIAP).

It provides a set of functions to call the on chip in application flash programming interface. User code executing from on chip flash or ram can call these function to erase and write the flash memory.

### 16.2 GFlash In Application Programming operation

[FLASHIAP\\_PrepareSectorForWrite\(\)](#) prepares a sector for write or erase operation.

[FLASHIAP\\_CopyRamToFlash\(\)](#) function programs the flash memory.

[FLASHIAP\\_EraseSector\(\)](#) function erase a flash sector. A sector must be erased before write operation.

### 16.3 Typical use case

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

#### Files

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

#### Typedefs

- typedef void(\* [IAP\\_ENTRY\\_T](#) )(uint32\_t cmd[5], uint32\_t stat[4])  
*IAP\_ENTRY API function type.*

### Enumerations

- enum `_flashiap_status` {  
    `kStatus_FLASHIAP_Success` = `kStatus_Success`,  
    `kStatus_FLASHIAP_InvalidCommand` = `MAKE_STATUS(kStatusGroup_FLASHIAP, 1U)`,  
    `kStatus_FLASHIAP_SrcAddrError`,  
    `kStatus_FLASHIAP_DstAddrError`,  
    `kStatus_FLASHIAP_SrcAddrNotMapped`,  
    `kStatus_FLASHIAP_DstAddrNotMapped`,  
    `kStatus_FLASHIAP_CountError`,  
    `kStatus_FLASHIAP_InvalidSector`,  
    `kStatus_FLASHIAP_SectorNotblank` = `MAKE_STATUS(kStatusGroup_FLASHIAP, 8U)`,  
    `kStatus_FLASHIAP_NotPrepared`,  
    `kStatus_FLASHIAP_CompareError`,  
    `kStatus_FLASHIAP_Busy`,  
    `kStatus_FLASHIAP_ParamError`,  
    `kStatus_FLASHIAP_AddrError` = `MAKE_STATUS(kStatusGroup_FLASHIAP, 13U)`,  
    `kStatus_FLASHIAP_AddrNotMapped`,  
    `kStatus_FLASHIAP_NoPower` = `MAKE_STATUS(kStatusGroup_FLASHIAP, 24U)`,  
    `kStatus_FLASHIAP_NoClock` }

*Flashiap status codes.*

- enum `_flashiap_commands` {  
    `kIapCmd_FLASHIAP_PrepareSectorforWrite` = 50U,  
    `kIapCmd_FLASHIAP_CopyRamToFlash` = 51U,  
    `kIapCmd_FLASHIAP_EraseSector` = 52U,  
    `kIapCmd_FLASHIAP_BlankCheckSector` = 53U,  
    `kIapCmd_FLASHIAP_ReadPartId` = 54U,  
    `kIapCmd_FLASHIAP_Read_BootromVersion` = 55U,  
    `kIapCmd_FLASHIAP_Compare` = 56U,  
    `kIapCmd_FLASHIAP_ReinvokeISP` = 57U,  
    `kIapCmd_FLASHIAP_ReadUid` = 58U,  
    `kIapCmd_FLASHIAP_ErasePage` = 59U,  
    `kIapCmd_FLASHIAP_ReadMisr` = 70U,  
    `kIapCmd_FLASHIAP_ReinvokeI2cSpiISP` = 71U }

*Flashiap command codes.*

### Functions

- static void `iap_entry` (uint32\_t \*cmd\_param, uint32\_t \*status\_result)  
*IAP\_ENTRY API function type.*
- `status_t FLASHIAP_PrepareSectorForWrite` (uint32\_t startSector, uint32\_t endSector)  
*Prepare sector for write operation.*
- `status_t FLASHIAP_CopyRamToFlash` (uint32\_t dstAddr, uint32\_t \*srcAddr, uint32\_t numOfBytes, uint32\_t systemCoreClock)  
*Copy RAM to flash.*
- `status_t FLASHIAP_EraseSector` (uint32\_t startSector, uint32\_t endSector, uint32\_t systemCoreClock)



- *Erase sector.*  
**status\_t FLASHIAP\_ErasePage** (uint32\_t startPage, uint32\_t endPage, uint32\_t systemCoreClock)  
*This function erases page(s).*
- **status\_t FLASHIAP\_BlankCheckSector** (uint32\_t startSector, uint32\_t endSector)  
*Blank check sector(s)*
- **status\_t FLASHIAP\_Compare** (uint32\_t dstAddr, uint32\_t \*srcAddr, uint32\_t numOfBytes)  
*Compare memory contents of flash with ram.*

## Driver version

- #define **FSL\_FLASHIAP\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 0))  
*Version 2.0.0.*

## 16.4 Macro Definition Documentation

### 16.4.1 #define FSL\_FLASHIAP\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

## 16.5 Enumeration Type Documentation

### 16.5.1 enum \_flashiap\_status

#### Enumerator

- kStatus\_FLASHIAP\_Success** Api is executed successfully.
- kStatus\_FLASHIAP\_InvalidCommand** Invalid command.
- kStatus\_FLASHIAP\_SrcAddrError** Source address is not on word boundary.
- kStatus\_FLASHIAP\_DstAddrError** Destination address is not on a correct boundary.
- kStatus\_FLASHIAP\_SrcAddrNotMapped** Source address is not mapped in the memory map.
- kStatus\_FLASHIAP\_DstAddrNotMapped** Destination address is not mapped in the memory map.
- kStatus\_FLASHIAP\_CountError** Byte count is not multiple of 4 or is not a permitted value.
- kStatus\_FLASHIAP\_InvalidSector** Sector number is invalid or end sector number is greater than start sector number.
- kStatus\_FLASHIAP\_SectorNotblank** One or more sectors are not blank.
- kStatus\_FLASHIAP\_NotPrepared** Command to prepare sector for write operation was not executed.
- kStatus\_FLASHIAP\_CompareError** Destination and source memory contents do not match.
- kStatus\_FLASHIAP\_Busy** Flash programming hardware interface is busy.
- kStatus\_FLASHIAP\_ParamError** Insufficient number of parameters or invalid parameter.
- kStatus\_FLASHIAP\_AddrError** Address is not on word boundary.
- kStatus\_FLASHIAP\_AddrNotMapped** Address is not mapped in the memory map.
- kStatus\_FLASHIAP\_NoPower** Flash memory block is powered down.
- kStatus\_FLASHIAP\_NoClock** Flash memory block or controller is not clocked.

## Function Documentation

### 16.5.2 enum \_flashiap\_commands

Enumerator

*kIapCmd\_FLASHIAP\_PrepareSectorforWrite* Prepare Sector for write.  
*kIapCmd\_FLASHIAP\_CopyRamToFlash* Copy RAM to flash.  
*kIapCmd\_FLASHIAP\_EraseSector* Erase Sector.  
*kIapCmd\_FLASHIAP\_BlankCheckSector* Blank check sector.  
*kIapCmd\_FLASHIAP\_ReadPartId* Read part id.  
*kIapCmd\_FLASHIAP\_Read\_BootromVersion* Read bootrom version.  
*kIapCmd\_FLASHIAP\_Compare* Compare.  
*kIapCmd\_FLASHIAP\_ReinvokeISP* Reinvoke ISP.  
*kIapCmd\_FLASHIAP\_ReadUid* Read Uid isp.  
*kIapCmd\_FLASHIAP\_ErasePage* Erase Page.  
*kIapCmd\_FLASHIAP\_ReadMisr* Read Misr.  
*kIapCmd\_FLASHIAP\_ReinvokeI2cSpiISP* Reinvoke I2C/SPI isp.

## 16.6 Function Documentation

### 16.6.1 static void iap\_entry ( uint32\_t \* cmd\_param, uint32\_t \* status\_result ) [inline], [static]

Wrapper for rom iap call

Parameters

<i>cmd_param</i>	IAP command and relevant parameter array.
<i>status_result</i>	IAP status result array.

Return values

<i>None.</i>	Status/Result is returned via status_result array.
--------------	--

### 16.6.2 status\_t FLASHIAP\_PrepareSectorForWrite ( uint32\_t startSector, uint32\_t endSector )

This function prepares sector(s) for write/erase operation. This function must be called before calling the [FLASHIAP\\_CopyRamToFlash\(\)](#) or [FLASHIAP\\_EraseSector\(\)](#) or [FLASHIAP\\_ErasePage\(\)](#) function. The end sector must be greater than or equal to start sector number.

## Parameters

<i>startSector</i>	Start sector number.
<i>endSector</i>	End sector number.

## Return values

<i>kStatus_FLASHIAP_Success</i>	Api was executed successfully.
<i>kStatus_FLASHIAP_NoPower</i>	Flash memory block is powered down.
<i>kStatus_FLASHIAP_NoClock</i>	Flash memory block or controller is not clocked.
<i>kStatus_FLASHIAP_InvalidSector</i>	Sector number is invalid or end sector number is greater than start sector number.
<i>kStatus_FLASHIAP_Busy</i>	Flash programming hardware interface is busy.

### 16.6.3 **status\_t FLASHIAP\_CopyRamToFlash ( uint32\_t dstAddr, uint32\_t \* srcAddr, uint32\_t numOfBytes, uint32\_t systemCoreClock )**

This function programs the flash memory. Corresponding sectors must be prepared via FLASHIAP\_PrepereSectorForWrite before calling calling this function. The addresses should be a 256 byte boundary and the number of bytes should be 256 | 512 | 1024 | 4096.

## Parameters

<i>dstAddr</i>	Destination flash address where data bytes are to be written.
<i>srcAddr</i>	Source ram address from where data bytes are to be read.
<i>numOfBytes</i>	Number of bytes to be written.
<i>systemCoreClock</i>	SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

## Return values

<i>kStatus_FLASHIAP_Success</i>	Api was executed successfully.
---------------------------------	--------------------------------

## Function Documentation

<i>kStatus_FLASHIAP_NoPower</i>	Flash memory block is powered down.
<i>kStatus_FLASHIAP_NoClock</i>	Flash memory block or controller is not clocked.
<i>kStatus_FLASHIAP_SrcAddrError</i>	Source address is not on word boundary.
<i>kStatus_FLASHIAP_DstAddrError</i>	Destination address is not on a correct boundary.
<i>kStatus_FLASHIAP_SrcAddrNotMapped</i>	Source address is not mapped in the memory map.
<i>kStatus_FLASHIAP_DstAddrNotMapped</i>	Destination address is not mapped in the memory map.
<i>kStatus_FLASHIAP_CountError</i>	Byte count is not multiple of 4 or is not a permitted value.
<i>kStatus_FLASHIAP_NotPrepared</i>	Command to prepare sector for write operation was not executed.
<i>kStatus_FLASHIAP_Busy</i>	Flash programming hardware interface is busy.

### 16.6.4 status\_t FLASHIAP\_EraseSector ( uint32\_t startSector, uint32\_t endSector, uint32\_t systemCoreClock )

This function erases sector(s). The end sector must be greater than or equal to start sector number. FLASHIAP\_PrepareSectorForWrite must be called before calling this function.

#### Parameters

<i>startSector</i>	Start sector number.
<i>endSector</i>	End sector number.
<i>systemCoreClock</i>	SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

#### Return values

<i>kStatus_FLASHIAP_Success</i>	Api was executed successfully.
---------------------------------	--------------------------------

<i>kStatus_FLASHIAP_No-Power</i>	Flash memory block is powered down.
<i>kStatus_FLASHIAP_No-Clock</i>	Flash memory block or controller is not clocked.
<i>kStatus_FLASHIAP_-InvalidSector</i>	Sector number is invalid or end sector number is greater than start sector number.
<i>kStatus_FLASHIAP_Not-Prepared</i>	Command to prepare sector for write operation was not executed.
<i>kStatus_FLASHIAP_Busy</i>	Flash programming hardware interface is busy.

### 16.6.5 status\_t FLASHIAP\_ErasePage ( uint32\_t startPage, uint32\_t endPage, uint32\_t systemCoreClock )

The end page must be greater than or equal to start page number. Corresponding sectors must be prepared via FLASHIAP\_PrepareSectorForWrite before calling this function.

Parameters

<i>startPage</i>	Start page number
<i>endPage</i>	End page number
<i>systemCore-Clock</i>	SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function.

Return values

<i>kStatus_FLASHIAP_-Success</i>	Api was executed successfully.
<i>kStatus_FLASHIAP_No-Power</i>	Flash memory block is powered down.
<i>kStatus_FLASHIAP_No-Clock</i>	Flash memory block or controller is not clocked.
<i>kStatus_FLASHIAP_-InvalidSector</i>	Page number is invalid or end page number is greater than start page number

## Function Documentation

<i>kStatus_FLASHIAP_Not-Prepared</i>	Command to prepare sector for write operation was not executed.
<i>kStatus_FLASHIAP_Busy</i>	Flash programming hardware interface is busy.

### 16.6.6 status\_t FLASHIAP\_BlankCheckSector ( uint32\_t startSector, uint32\_t endSector )

Blank check single or multiples sectors of flash memory. The end sector must be greater than or equal to start sector number. It can be used to verify the sector eraseure after FLASHIAP\_EraseSector call.

Parameters

<i>startSector</i>	: Start sector number. Must be greater than or equal to start sector number
<i>endSector</i>	: End sector number

Return values

<i>kStatus_FLASHIAP_Success</i>	One or more sectors are in erased state.
<i>kStatus_FLASHIAP_No-Power</i>	Flash memory block is powered down.
<i>kStatus_FLASHIAP_No-Clock</i>	Flash memory block or controller is not clocked.
<i>kStatus_FLASHIAP_SectorNotblank</i>	One or more sectors are not blank.

### 16.6.7 status\_t FLASHIAP\_Compare ( uint32\_t dstAddr, uint32\_t \* srcAddr, uint32\_t numBytes )

This function compares the contents of flash and ram. It can be used to verify the flash memory contents after FLASHIAP\_CopyRamToFlash call.

Parameters

<i>dstAddr</i>	Destination flash address.
----------------	----------------------------

<i>srcAddr</i>	Source ram address.
<i>numOfBytes</i>	Number of bytes to be compared.

## Return values

<i>kStatus_FLASHIAP_Success</i>	Contents of flash and ram match.
<i>kStatus_FLASHIAP_NoPower</i>	Flash memory block is powered down.
<i>kStatus_FLASHIAP_NoClock</i>	Flash memory block or controller is not clocked.
<i>kStatus_FLASHIAP_AddrError</i>	Address is not on word boundary.
<i>kStatus_FLASHIAP_AddrNotMapped</i>	Address is not mapped in the memory map.
<i>kStatus_FLASHIAP_CountError</i>	Byte count is not multiple of 4 or is not a permitted value.
<i>kStatus_FLASHIAP_CompareError</i>	Destination and source memory contents do not match.





## Chapter 17

# UTICK: MictoTick Timer Driver

### 17.1 Overview

The MCUXpresso SDK provides Peripheral driver for the UTICK module of MCUXpresso SDK devices. UTICK driver is created to help user to operate the UTICK module. The UTICK timer can be used as a low power timer. The APIs can be used to enable the UTICK module, initialize it and set the time. UTICK can be used as a wake up source from low power mode.

### 17.2 Typical use case

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

#### Files

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

#### Typedefs

- typedef void(\* [utick\\_callback\\_t](#))(void)  
*UTICK callback function.*

#### Enumerations

- enum [utick\\_mode\\_t](#) {  
    [kUTICK\\_Onetime](#) = 0x0U,  
    [kUTICK\\_Repeat](#) = 0x1U }  
*UTICK timer operational mode.*

#### Driver version

- #define [FSL\\_UTICK\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 0))  
*UTICK driver version 2.0.0.*

#### Initialization and deinitialization

- void [UTICK\\_Init](#) (UTICK\_Type \*base)  
*Initializes an UTICK by turning its bus clock on.*
- void [UTICK\\_Deinit](#) (UTICK\_Type \*base)  
*Deinitializes a UTICK instance.*
- uint32\_t [UTICK\\_GetStatusFlags](#) (UTICK\_Type \*base)  
*Get Status Flags.*
- void [UTICK\\_ClearStatusFlags](#) (UTICK\_Type \*base)  
*Clear Status Interrupt Flags.*

## Function Documentation

- void [UTICK\\_SetTick](#) (UTICK\_Type \*base, [utick\\_mode\\_t](#) mode, uint32\_t count, [utick\\_callback\\_t](#) cb)  
*Starts UTICK.*
- void [UTICK\\_HandleIRQ](#) (UTICK\_Type \*base, [utick\\_callback\\_t](#) cb)  
*UTICK Interrupt Service Handler.*

## 17.3 Macro Definition Documentation

### 17.3.1 #define FSL\_UTICK\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

## 17.4 Typedef Documentation

### 17.4.1 typedef void(\* utick\_callback\_t)(void)

## 17.5 Enumeration Type Documentation

### 17.5.1 enum utick\_mode\_t

Enumerator

*kUTICK\_Onetime* Trigger once.  
*kUTICK\_Repeat* Trigger repeatedly.

## 17.6 Function Documentation

### 17.6.1 void UTICK\_Init ( UTICK\_Type \* *base* )

### 17.6.2 void UTICK\_Deinit ( UTICK\_Type \* *base* )

This function shuts down Utick bus clock

Parameters

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

### 17.6.3 uint32\_t UTICK\_GetStatusFlags ( UTICK\_Type \* *base* )

This returns the status flag

Parameters

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

Returns

status register value

#### 17.6.4 void UTICK\_ClearStatusFlags ( UTICK\_Type \* *base* )

This clears intr status flag

Parameters

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

Returns

none

#### 17.6.5 void UTICK\_SetTick ( UTICK\_Type \* *base*, utick\_mode\_t *mode*, uint32\_t *count*, utick\_callback\_t *cb* )

This function starts a repeat/onetime countdown with an optional callback

Parameters

<i>base</i>	UTICK peripheral base address.
<i>mode</i>	UTICK timer mode (ie kUTICK_onetime or kUTICK_repeat)
<i>count</i>	UTICK timer mode (ie kUTICK_onetime or kUTICK_repeat)
<i>cb</i>	UTICK callback (can be left as NULL if none, otherwise should be a void func(void))

Returns

none

#### 17.6.6 void UTICK\_HandleIRQ ( UTICK\_Type \* *base*, utick\_callback\_t *cb* )

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in [UTICK\\_SetTick\(\)](#)). if no user callback is scheduled, the interrupt will simply be cleared.

## Function Documentation

### Parameters

<i>base</i>	UTICK peripheral base address.
<i>cb</i>	callback scheduled for this instance of UTICK

### Returns

none

## Chapter 18

# FMEAS: Frequency Measure Driver

### 18.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Frequency Measure function of MCUXpresso SDK devices' SYSCON module.

It measures frequency of any on-chip or off-chip clock signal. The more precise and higher accuracy clock is selected as a reference clock. The resulting frequency is internally computed from the ratio of value of selected target and reference clock counters.

### 18.2 Frequency Measure Driver operation

[INPUTMUX\\_AttachSignal\(\)](#) function has to be used to select reference and target clock signal sources.

[FMEAS\\_StartMeasure\(\)](#) function starts the measurement cycle.

[FMEAS\\_IsMeasureComplete\(\)](#) can be polled to check if the measurement cycle has finished.

[FMEAS\\_GetFrequency\(\)](#) returns the frequency of the target clock. Frequency of the reference clock has to be provided as a parameter.

### 18.3 Typical use case

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

#### Files

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

#### Driver version

- #define [FSL\\_FMEAS\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 0))  
*Defines LPC Frequency Measure driver version 2.0.0.*

#### FMEAS Functional Operation

- static void [FMEAS\\_StartMeasure](#) (SYSCON\_Type \*base)  
*Starts a frequency measurement cycle.*
- static bool [FMEAS\\_IsMeasureComplete](#) (SYSCON\_Type \*base)  
*Indicates when a frequency measurement cycle is complete.*
- uint32\_t [FMEAS\\_GetFrequency](#) (SYSCON\_Type \*base, uint32\_t refClockRate)  
*Returns the computed value for a frequency measurement cycle.*

## Function Documentation

### 18.4 Macro Definition Documentation

#### 18.4.1 #define FSL\_FMEAS\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

Change log:

- Version 2.0.0
  - initial version

### 18.5 Function Documentation

#### 18.5.1 static void FMEAS\_StartMeasure ( SYSCON\_Type \* *base* ) [inline], [static]

Parameters

<i>base</i>	: SYSCON peripheral base address.
-------------	-----------------------------------

#### 18.5.2 static bool FMEAS\_IsMeasureComplete ( SYSCON\_Type \* *base* ) [inline], [static]

Parameters

<i>base</i>	: SYSCON peripheral base address.
-------------	-----------------------------------

Returns

true if a measurement cycle is active, otherwise false.

#### 18.5.3 uint32\_t FMEAS\_GetFrequency ( SYSCON\_Type \* *base*, uint32\_t *refClockRate* )

Parameters

<i>base</i>	: SYSCON peripheral base address.
-------------	-----------------------------------

<i>refClockRate</i>	: Reference clock rate used during the frequency measurement cycle.
---------------------	---

Returns

Frequency in Hz.





## Chapter 19

# CRC: Cyclic Redundancy Check Driver

### 19.1 Overview

MCUXpresso SDK provides the 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 provides three variants of polynomials, a programmable seed and other parameters required to implement a 16-bit or 32-bit CRC standard.

### 19.2 CRC Driver Initialization and Configuration

[CRC\\_Init\(\)](#) function enables the clock for the CRC module in the LPC SYSCON block and fully (re-)configures the CRC module according to configuration structure. It also starts checksum computation by writing the seed.

The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting new checksum computation, the seed shall be set to the initial checksum per the CRC protocol specification. For continued checksum operation, the seed shall be set to the intermediate checksum value as obtained from previous calls to [CRC\\_GetConfig\(\)](#) function. After [CRC\\_Init\(\)](#), one or multiple [CRC\\_WriteData\(\)](#) calls follow to update checksum with data, then [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#) follows to read the result. [CRC\\_Init\(\)](#) can be called as many times as required, thus, allows for runtime changes of CRC protocol.

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

[CRC\\_Deinit\(\)](#) function disables clock to the CRC module.

[CRC\\_Reset\(\)](#) performs hardware reset of the CRC module.

### 19.3 CRC Write Data

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

### 19.4 CRC Get Checksum

The [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#) function is used to read the CRC module checksum register. The bit reverse and 1's complement operations are already applied to the result if previously configured. Use [CRC\\_GetConfig\(\)](#) function to get the actual checksum without bit reverse and 1's complement applied so it can be used as seed when resuming calculation later.

## Comments about API usage in RTOS

[CRC\\_Init\(\)](#) / [CRC\\_WriteData\(\)](#) / [CRC\\_Get16bitResult\(\)](#) to get final checksum.

[CRC\\_Init\(\)](#) / [CRC\\_WriteData\(\)](#) / ... / [CRC\\_WriteData\(\)](#) / [CRC\\_Get16bitResult\(\)](#) to get final checksum.

[CRC\\_Init\(\)](#) / [CRC\\_WriteData\(\)](#) / [CRC\\_GetConfig\(\)](#) to get intermediate checksum to be used as seed value in future.

[CRC\\_Init\(\)](#) / [CRC\\_WriteData\(\)](#) / ... / [CRC\\_WriteData\(\)](#) / [CRC\\_GetConfig\(\)](#) to get intermediate checksum.

## 19.5 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\(\)](#) or [CRC\\_GetConfig\(\)](#)

shall be protected by RTOS mutex to protect CRC module against concurrent accesses from different tasks.

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/crcRefer 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/crcRefer 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

## Files

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

## Data Structures

- struct [crc\\_config\\_t](#)  
*CRC protocol configuration. [More...](#)*

## Macros

- #define [CRC\\_DRIVER\\_USE\\_CRC16\\_CCITT\\_FALSE\\_AS\\_DEFAULT](#) 1  
*Default configuration structure filled by [CRC\\_GetDefaultConfig\(\)](#).*

## Enumerations

- enum [crc\\_polynomial\\_t](#) {  
    [kCRC\\_Polynomial\\_CRC\\_CCITT](#) = 0U,  
    [kCRC\\_Polynomial\\_CRC\\_16](#) = 1U,  
    [kCRC\\_Polynomial\\_CRC\\_32](#) = 2U }  
*CRC polynomials to use.*

## 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\\_Reset](#) (CRC\_Type \*base)  
*resets CRC peripheral module.*
- void [CRC\\_GetDefaultConfig](#) ([crc\\_config\\_t](#) \*config)  
*Loads default values to CRC protocol configuration structure.*
- void [CRC\\_GetConfig](#) (CRC\_Type \*base, [crc\\_config\\_t](#) \*config)  
*Loads actual values configured in CRC peripheral to CRC protocol configuration structure.*
- void [CRC\\_WriteData](#) (CRC\_Type \*base, const uint8\_t \*data, size\_t dataSize)  
*Writes data to the CRC module.*
- static uint32\_t [CRC\\_Get32bitResult](#) (CRC\_Type \*base)  
*Reads 32-bit checksum from the CRC module.*
- static uint16\_t [CRC\\_Get16bitResult](#) (CRC\_Type \*base)  
*Reads 16-bit checksum from the CRC module.*

## Driver version

- #define [FSL\\_CRC\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 1))  
*CRC driver version.*

## 19.6 Data Structure Documentation

### 19.6.1 struct [crc\\_config\\_t](#)

This structure holds the configuration for the CRC protocol.

## Data Fields

- [crc\\_polynomial\\_t](#) [polynomial](#)  
*CRC polynomial.*
- bool [reverseIn](#)  
*Reverse bits on input.*
- bool [complementIn](#)  
*Perform 1's complement on input.*
- bool [reverseOut](#)  
*Reverse bits on output.*
- bool [complementOut](#)  
*Perform 1's complement on output.*
- uint32\_t [seed](#)  
*Starting checksum value.*

## Function Documentation

### 19.6.1.0.0.26 Field Documentation

19.6.1.0.0.26.1 `crc_polynomial_t crc_config_t::polynomial`

19.6.1.0.0.26.2 `bool crc_config_t::reverseIn`

19.6.1.0.0.26.3 `bool crc_config_t::complementIn`

19.6.1.0.0.26.4 `bool crc_config_t::reverseOut`

19.6.1.0.0.26.5 `bool crc_config_t::complementOut`

19.6.1.0.0.26.6 `uint32_t crc_config_t::seed`

## 19.7 Macro Definition Documentation

19.7.1 `#define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

Version 2.0.1.

Current version: 2.0.1

Change log:

- Version 2.0.0
  - initial version
- Version 2.0.1
  - add explicit type cast when writing to WR\_DATA

19.7.2 `#define CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT 1`

Uses CRC-16/CCITT-FALSE as default.

## 19.8 Enumeration Type Documentation

19.8.1 `enum crc_polynomial_t`

Enumerator

*kCRC\_Polynomial\_CRC\_CCITT*  $x^{16}+x^{12}+x^5+1$

*kCRC\_Polynomial\_CRC\_16*  $x^{16}+x^{15}+x^2+1$

*kCRC\_Polynomial\_CRC\_32*  $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

## 19.9 Function Documentation

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

This functions enables the CRC peripheral clock in the LPC SYSCON block. It also configures the CRC engine and starts checksum computation by writing the seed.

## Parameters

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

**19.9.2 static void CRC\_Deinit ( CRC\_Type \* *base* ) [inline], [static]**

This functions disables the CRC peripheral clock in the LPC SYSCON block.

## Parameters

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

**19.9.3 void CRC\_Reset ( CRC\_Type \* *base* )**

## Parameters

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

**19.9.4 void CRC\_GetDefaultConfig ( crc\_config\_t \* *config* )**

Loads default values to CRC protocol configuration structure. The default values are:

```
* config->polynomial = kCRC_Polynomial_CRC_CCITT;
* config->reverseIn = false;
* config->complementIn = false;
* config->reverseOut = false;
* config->complementOut = false;
* config->seed = 0xFFFFU;
*
```

## Parameters

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

**19.9.5 void CRC\_GetConfig ( CRC\_Type \* *base*, crc\_config\_t \* *config* )**

The values, including seed, can be used to resume CRC calculation later.

## Function Documentation

### Parameters

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

### 19.9.6 void CRC\_WriteData ( CRC\_Type \* *base*, const uint8\_t \* *data*, size\_t *dataSize* )

Writes input data buffer bytes to CRC data register.

### Parameters

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

### 19.9.7 static uint32\_t CRC\_Get32bitResult ( CRC\_Type \* *base* ) [inline], [static]

Reads CRC data register.

### Parameters

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

### Returns

final 32-bit checksum, after configured bit reverse and complement operations.

### 19.9.8 static uint16\_t CRC\_Get16bitResult ( CRC\_Type \* *base* ) [inline], [static]

Reads CRC data register.

### Parameters

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

## Returns

final 16-bit checksum, after configured bit reverse and complement operations.





## Chapter 20

# INPUTMUX: Input Multiplexing Driver

### 20.1 Overview

The MCUXpresso SDK provides a driver for the Input multiplexing (INPUTMUX).

It configures the inputs to the pin interrupt block, DMA trigger and the frequency measure function. Once configured the clock is not needed for the inputmux.

### 20.2 Input Multiplexing Driver operation

INPUTMUX\_AttachSignal function configures the specified input

### 20.3 Typical use case

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

### Files

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

### Functions

- void [INPUTMUX\\_Init](#) (INPUTMUX\_Type \*base)  
*Initialize INPUTMUX peripheral.*
- void [INPUTMUX\\_AttachSignal](#) (INPUTMUX\_Type \*base, uint32\_t index, inputmux\_connection\_t connection)  
*Attaches a signal.*
- void [INPUTMUX\\_Deinit](#) (INPUTMUX\_Type \*base)  
*Deinitialize INPUTMUX peripheral.*

### Driver version

- #define [FSL\\_INPUTMUX\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 0))  
*Group interrupt driver version for SDK.*

### 20.4 Macro Definition Documentation

#### 20.4.1 #define FSL\_INPUTMUX\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

Version 2.0.0.

### 20.5 Function Documentation

#### 20.5.1 void INPUTMUX\_Init ( INPUTMUX\_Type \* *base* )

This function enables the INPUTMUX clock.

## Parameters

<i>base</i>	Base address of the INPUTMUX peripheral.
-------------	--

## Return values

<i>None.</i>	
--------------	--

### 20.5.2 void INPUTMUX\_AttachSignal ( INPUTMUX\_Type \* *base*, uint32\_t *index*, inputmux\_connection\_t *connection* )

This function gates the INPUTMUX clock.

## Parameters

<i>base</i>	Base address of the INPUTMUX peripheral.
<i>index</i>	Destination peripheral to attach the signal to.
<i>connection</i>	Selects connection.

## Return values

<i>None.</i>	
--------------	--

### 20.5.3 void INPUTMUX\_Deinit ( INPUTMUX\_Type \* *base* )

This function disables the INPUTMUX clock.

## Parameters

<i>base</i>	Base address of the INPUTMUX peripheral.
-------------	--

## Return values

<i>None.</i>	
--------------	--



## Chapter 21

# IOCON: I/O pin configuration

### 21.1 Overview

The MCUXpresso SDK provides Peripheral driver for the I/O pin configuration (IOCON) module of MCUXpresso SDK devices.

### 21.2 Function groups

#### 21.2.1 Pin mux set

The function `IOCONPinMuxSet()` set pinmux for single pin according to selected configuration.

#### 21.2.2 Pin mux set

The function [IOCON\\_SetPinMuxing\(\)](#) set pinmux for group of pins according to selected configuration.

### 21.3 Typical use case

Example use of IOCON API to selection of GPIO mode. Refer to the driver examples codes located at `<SDK_ROOT>/boards/<BOARD>/driver_examples/iocon`

### Files

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

### Data Structures

- struct [iocon\\_group\\_t](#)  
*Array of IOCON pin definitions passed to [IOCON\\_SetPinMuxing\(\)](#) must be in this format. [More...](#)*

### Macros

- #define [IOCON\\_FUNC0](#) 0x0  
*IOCON function and mode selection definitions.*
- #define [IOCON\\_FUNC1](#) 0x1  
*Selects pin function 1.*
- #define [IOCON\\_FUNC2](#) 0x2  
*Selects pin function 2.*
- #define [IOCON\\_FUNC3](#) 0x3  
*Selects pin function 3.*
- #define [IOCON\\_FUNC4](#) 0x4  
*Selects pin function 4.*

## Typical use case

- #define `IOCON_FUNC5` 0x5  
*Selects pin function 5.*
- #define `IOCON_FUNC6` 0x6  
*Selects pin function 6.*
- #define `IOCON_FUNC7` 0x7  
*Selects pin function 7.*
- #define `IOCON_MODE_INACT` (0x0 << 3)  
*No additional pin function.*
- #define `IOCON_MODE_PULLDOWN` (0x1 << 3)  
*Selects pull-down function.*
- #define `IOCON_MODE_PULLUP` (0x2 << 3)  
*Selects pull-up function.*
- #define `IOCON_MODE_REPEATER` (0x3 << 3)  
*Selects pin repeater function.*
- #define `IOCON_HYS_EN` (0x1 << 5)  
*Enables hysteresis.*
- #define `IOCON_GPIO_MODE` (0x1 << 5)  
*GPIO Mode.*
- #define `IOCON_I2C_SLEW` (0x0 << 5)  
*I2C Slew Rate Control.*
- #define `IOCON_INV_EN` (0x1 << 6)  
*Enables invert function on input.*
- #define `IOCON_ANALOG_EN` (0x0 << 7)  
*Enables analog function by setting 0 to bit 7.*
- #define `IOCON_DIGITAL_EN` (0x1 << 7)  
*Enables digital function by setting 1 to bit 7 (default).*
- #define `IOCON_STDI2C_EN` (0x1 << 8)  
*I2C standard mode/fast-mode.*
- #define `IOCON_FASTI2C_EN` (0x3 << 8)  
*I2C Fast-mode Plus and high-speed slave.*
- #define `IOCON_INPFILT_OFF` (0x1 << 8)  
*Input filter Off for GPIO pins.*
- #define `IOCON_INPFILT_ON` (0x0 << 8)  
*Input filter On for GPIO pins.*
- #define `IOCON_OPENDRAIN_EN` (0x1 << 10)  
*Enables open-drain function.*
- #define `IOCON_S_MODE_0CLK` (0x0 << 11)  
*Bypass input filter.*
- #define `IOCON_S_MODE_1CLK` (0x1 << 11)  
*Input pulses shorter than 1 filter clock are rejected.*
- #define `IOCON_S_MODE_2CLK` (0x2 << 11)  
*Input pulses shorter than 2 filter clock2 are rejected.*
- #define `IOCON_S_MODE_3CLK` (0x3 << 11)  
*Input pulses shorter than 3 filter clock2 are rejected.*
- #define `IOCON_S_MODE(clks)` ((clks) << 11)  
*Select clocks for digital input filter mode.*
- #define `IOCON_CLKDIV(div)` ((div) << 13)  
*Select peripheral clock divider for input filter sampling clock,  $2^n$ ,  $n=0-6$ .*

## Functions

- `__STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type *base, uint8_t port, uint8_t pin, uint32_t modefunc)`  
*Sets I/O Control pin mux.*
- `__STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type *base, const iocon_group_t *pin-Array, uint32_t arrayLength)`  
*Set all I/O Control pin muxing.*

## Driver version

- `#define LPC_IOCON_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`  
*IOCON driver version 2.0.0.*

## 21.4 Data Structure Documentation

### 21.4.1 struct iocon\_group\_t

## 21.5 Macro Definition Documentation

### 21.5.1 #define LPC\_IOCON\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

### 21.5.2 #define IOCON\_FUNC0 0x0

#### Note

See the User Manual for specific modes and functions supported by the various pins. Selects pin function 0

## 21.6 Function Documentation

### 21.6.1 `__STATIC_INLINE void IOCON_PinMuxSet ( IOCON_Type * base, uint8_t port, uint8_t pin, uint32_t modefunc )`

#### Parameters

<i>base</i>	: The base of IOCON peripheral on the chip
<i>port</i>	: GPIO port to mux
<i>pin</i>	: GPIO pin to mux
<i>modefunc</i>	: OR'ed values of type IOCON_*

#### Returns

Nothing

**21.6.2** `__STATIC_INLINE void IOCON_SetPinMuxing ( IOCON_Type * base, const iocon_group_t * pinArray, uint32_t arrayLength )`



#### Parameters

<i>base</i>	: The base of IOCON peripheral on the chip
<i>pinArray</i>	: Pointer to array of pin mux selections
<i>arrayLength</i>	: Number of entries in pinArray

#### Returns

Nothing





## Chapter 22

# **SYSCON: System Configuration**

### **22.1 Overview**

The MCUXpresso SDK provides a peripheral clock and power driver for the SYSCON module of MCU-Xpresso SDK devices. For further details, see corresponding chapter.

#### **Modules**

- [Clock driver](#)
- [Power driver](#)

## 22.2 Clock driver

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

### 22.2.1 Function description

Clock driver provides these functions:

- Functions to initialize the Core clock to given frequency
- Functions to configure the clock selection muxes.
- Functions to setup peripheral clock dividers
- Functions to set the flash wait states for the input frequency
- Functions to get the frequency of the selected clock
- Functions to set PLL frequency

#### 22.2.1.1 SYSCON Clock frequency functions

SYSCON clock module provides clocks, such as MCLKCLK, ADCCLK, DMICCLK, MCGFLLCLK, FXCOMCLK, WDTOSC, RTCOSC, USBCLK and SYSPLL. The functions `CLOCK_EnableClock()` and `CLOCK_DisableClock()` enables and disables the various clocks. `CLOCK_SetupFROClocking()` initializes the FRO to 12MHz, 48 MHz or 96 MHz frequency. `CLOCK_SetupPLLData()`, `CLOCK_SetupSystemPLLPrec()`, and `CLOCK_SetPLLFreq()` functions are used to setup the PLL. The SYSCON clock driver provides functions to get the frequency of these clocks, such as `CLOCK_GetFreq()`, `CLOCK_GetFro12MFreq()`, `CLOCK_GetExtClkFreq()`, `CLOCK_GetWdtOscFreq()`, `CLOCK_GetFroHfFreq()`, `CLOCK_GetPllOutFreq()`, `CLOCK_GetOsc32KFreq()`, `CLOCK_GetCoreSysClkFreq()`, `CLOCK_GetI2SMClkFreq()`, `CLOCK_GetFlexCommClkFreq` and `CLOCK_GetAsyncApbClkFreq`.

#### 22.2.1.2 SYSCON clock Selection Muxes

The SYSCON clock driver provides the function to configure the clock selected. The function `CLOCK_AttachClk()` is implemented for this. The function selects the clock source for a particular peripheral like MAINCLK, DMIC, FLEXCOMM, USB, ADC and PLL.

#### 22.2.1.3 SYSCON clock dividers

The SYSCON clock module provides the function to setup the peripheral clock dividers. The function `CLOCK_SetClkDiv()` configures the CLKDIV registers for various peripherals like USB, DMIC, I2S, SYSTICK, AHB, ADC and also for CLKOUT and TRACE functions.

#### 22.2.1.4 SYSCON flash wait states

The SYSCON clock driver provides the function `CLOCK_SetFLASHAccessCyclesForFreq()` that configures FLASHCFG register with a selected FLASHTIM value.

#### 22.2.2 Typical use case

```
POWER_DisablePD(kPDRUNCFG_PD_FRO_EN); /*!< Ensure FRO is on so that we can switch to its 12MHz
```

### 22.3 Power driver

The MCUXpresso SDK provides a power driver for the MCUXpresso SDK devices.

#### 22.3.1 Function description

Power driver and library provides these functions:

- Functions to enable and disable power to different peripherals
- Functions to enable and disable deep sleep in the ARM Core.
- Functions to enter deep sleep mode and deep power down mode
- Functions to set the voltages for different frequency for both normal regulation and low power regulation modes

##### 22.3.1.1 Power enable and disable

Power driver provides two API's `POWER_EnablePD()` and `POWER_DisablePD()` to enable or disable the `PDRUNCFG` bits in `SYSCON`. The `PDRUNCFG` has an inverted logic, the peripheral is powered on when the bit is cleared and powered off when bit is set. So the API `POWER_DisablePD()` is used to power on a peripheral and `POWER_EnablePD()` is used to power off a peripheral. The API's take a parameter of type `pd_bit_t` which organizes the `PDRUNCFG` bits. The driver also provides two separate API's to power down and power up Flash, `POWER_PowerDownFlash()` and `POWER_PowerUpFlash()`

##### 22.3.1.2 Enable and Disable Deep Sleep in Core

The power driver provides two API's `POWER_EnableDeepSleep()` and `POWER_DisableDeepSleep()` to enable or disable the deep sleep bit in the ARM Core. `POWER_EnableDeepSleep()` is used to enable deep sleep and `POWER_DisableDeepSleep()` is used to disable deep sleep.

##### 22.3.1.3 Entering Power Modes

The Power library provides two API's to enter low power modes, Deep Sleep and Deep Power Down. Deep Sleep is a sleep mode in which the ARM Core, Flash and many other peripheral are turned off to save power. The processor can be woken by an IO activity and will resume executing from next instruction after sleep. If a peripheral or RAM needs to On for wakeup or to retain memory then those peripheral need to be kept on during deep sleep. Deep power down is a power down mode where the processor resets upon wake up and during power down the entire part is powered down except for the RTC. For Deep Power Down only the Reset and RTC Alarm or WakeUp can be wakeup sources. The power library provides an API `POWER_EnterDeepSleep()` to enter deep sleep mode. This function takes a parameter which is a bit mask of the `PDRUNCFG` register. Any bit that is set will be powered on during deep sleep. So this mask would usually has the RAM memory that needs to retain power and also any wakeup source. The API `POWER_EnterDeepPowerDown()` is used to enter deep power down mode. This API also has a parameter but since the voltage is cut off for the peripheral this parameter has no effect

#### 22.3.1.4 Set Voltages for Frequency

The power library provides API's to set the voltage for the desired operating frequency of the processor. The voltage regulation system can be in normal regulation mode or in low power regulation mode. The API `POWER_SetVoltageForFreq()` is used to set the voltage for normal regulation mode. Based on the frequency parameter the optimum voltage level is set. The API `POWER_SetLowPowerVoltageForFreq()` is used to set the low power voltage regulation mode and set the voltages for the desired frequency. For `POWER_SetLowPowerVoltageForFreq()` only two FRO frequencies are supported, 12MHz and 48MHz.

### 22.3.2 Typical use case

#### 22.3.2.1 Power Enable and Set Voltage example

```
POWER_DisablePD(kPDRUNCFG_PD_FRO_EN); /*!< Ensure FRO is on so that we can switch to its 12MHz
```





## Chapter 23

# GPIO: General Purpose I/O

### 23.1 Overview

The MCUXpresso SDK provides Peripheral driver for the General Purpose I/O (GPIO) module of MCU-Xpresso SDK devices.

### 23.2 Function groups

#### 23.2.1 Initialization and deinitialization

The function [GPIO\\_PinInit\(\)](#) initializes the GPIO with specified configuration.

#### 23.2.2 Pin manipulation

The function [GPIO\\_WritePinOutput\(\)](#) set output state of selected GPIO pin. The function [GPIO\\_ReadPinInput\(\)](#) read input value of selected GPIO pin.

#### 23.2.3 Port manipulation

The function [GPIO\\_SetPinsOutput\(\)](#) sets the output level of selected GPIO pins to the logic 1. The function [GPIO\\_ClearPinsOutput\(\)](#) sets the output level of selected GPIO pins to the logic 0. The function [GPIO\\_TogglePinsOutput\(\)](#) reverse the output level of selected GPIO pins. The function [GPIO\\_ReadPinsInput\(\)](#) read input value of selected port.

#### 23.2.4 Port masking

The function [GPIO\\_SetPortMask\(\)](#) set port mask, only pins masked by 0 will be enabled in following functions. The function [GPIO\\_WriteMPort\(\)](#) sets the state of selected GPIO port, only pins masked by 0 will be affected. The function [GPIO\\_ReadMPort\(\)](#) reads the state of selected GPIO port, only pins masked by 0 are enabled for read, pins masked by 1 are read as 0.

### 23.3 Typical use case

Example use of GPIO API. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/gpio

## Typical use case

## Files

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

## 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 }  
*LPC GPIO direction definition.*

## Functions

- static void [GPIO\\_PortSet](#) (GPIO\_Type \*base, uint32\_t port, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 1.*
- static void [GPIO\\_SetPinsOutput](#) (GPIO\_Type \*base, uint32\_t port, 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 port, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 0.*
- static void [GPIO\\_ClearPinsOutput](#) (GPIO\_Type \*base, uint32\_t port, 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 port, uint32\_t mask)  
*Reverses current output logic of the multiple GPIO pins.*
- static void [GPIO\\_TogglePinsOutput](#) (GPIO\_Type \*base, uint32\_t port, uint32\_t mask)  
*Reverses current output logic of the multiple GPIO pins.*

## Driver version

- #define [FSL\\_GPIO\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 1, 1))  
*LPC GPIO driver version 2.1.1.*

## GPIO Configuration

- void [GPIO\\_PortInit](#) (GPIO\_Type \*base, uint32\_t port)  
*Initializes the GPIO peripheral.*
- static void [GPIO\\_Init](#) (GPIO\_Type \*base, uint32\_t port)  
*Initializes the GPIO peripheral.*
- void [GPIO\\_PinInit](#) (GPIO\_Type \*base, uint32\_t port, 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 port, uint32\_t pin, uint8\_t output)  
*Sets the output level of the one GPIO pin to the logic 1 or 0.*

- static void [GPIO\\_WritePinOutput](#) (GPIO\_Type \*base, uint32\_t port, uint32\_t pin, uint8\_t output)  
*Sets the output level of the one GPIO pin to the logic 1 or 0.*

## GPIO Input Operations

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

## 23.4 Data Structure Documentation

### 23.4.1 struct gpio\_pin\_config\_t

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

#### Data Fields

- [gpio\\_pin\\_direction\\_t pinDirection](#)  
*GPIO direction, input or output.*
- uint8\_t [outputLogic](#)  
*Set default output logic, no use in input.*

## 23.5 Macro Definition Documentation

### 23.5.1 #define FSL\_GPIO\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 1))

## 23.6 Enumeration Type Documentation

### 23.6.1 enum gpio\_pin\_direction\_t

Enumerator

*kGPIO\_DigitalInput* Set current pin as digital input.  
*kGPIO\_DigitalOutput* Set current pin as digital output.

## 23.7 Function Documentation

### 23.7.1 void GPIO\_PortInit ( GPIO\_Type \* *base*, uint32\_t *port* )

This function ungates the GPIO clock.

## Function Documentation

### Parameters

<i>base</i>	GPIO peripheral base pointer.
<i>port</i>	GPIO port number.

**23.7.2 static void GPIO\_Init ( GPIO\_Type \* *base*, uint32\_t *port* ) [inline], [static]**

**23.7.3 void GPIO\_PinInit ( GPIO\_Type \* *base*, uint32\_t *port*, uint32\_t *pin*, const gpio\_pin\_config\_t \* *config* )**

To initialize the GPIO, define a pin configuration, either input or output, in the user file. Then, call the [GPIO\\_PinInit\(\)](#) function.

This is an example to define an input pin or 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(Typically GPIO)
<i>port</i>	GPIO port number
<i>pin</i>	GPIO pin number
<i>config</i>	GPIO pin configuration pointer

**23.7.4 static void GPIO\_PinWrite ( GPIO\_Type \* *base*, uint32\_t *port*, uint32\_t *pin*, uint8\_t *output* ) [inline], [static]**

## Parameters

<i>base</i>	GPIO peripheral base pointer(Typically GPIO)
<i>port</i>	GPIO port number
<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>

**23.7.5** `static void GPIO_WritePinOutput ( GPIO_Type * base, uint32_t port,  
uint32_t pin, uint8_t output ) [inline], [static]`

**23.7.6** `static uint32_t GPIO_PinRead ( GPIO_Type * base, uint32_t port, uint32_t  
pin ) [inline], [static]`

## Parameters

<i>base</i>	GPIO peripheral base pointer(Typically GPIO)
<i>port</i>	GPIO port number
<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>
-------------	--

**23.7.7** `static uint32_t GPIO_ReadPinInput ( GPIO_Type * base, uint32_t port,  
uint32_t pin ) [inline], [static]`

**23.7.8** `static void GPIO_PortSet ( GPIO_Type * base, uint32_t port, uint32_t mask  
) [inline], [static]`

## Function Documentation

### Parameters

<i>base</i>	GPIO peripheral base pointer(Typically GPIO)
<i>port</i>	GPIO port number
<i>mask</i>	GPIO pin number macro

**23.7.9** `static void GPIO_SetPinsOutput ( GPIO_Type * base, uint32_t port,  
uint32_t mask ) [inline], [static]`

**23.7.10** `static void GPIO_PortClear ( GPIO_Type * base, uint32_t port, uint32_t  
mask ) [inline], [static]`

### Parameters

<i>base</i>	GPIO peripheral base pointer(Typically GPIO)
<i>port</i>	GPIO port number
<i>mask</i>	GPIO pin number macro

**23.7.11** `static void GPIO_ClearPinsOutput ( GPIO_Type * base, uint32_t port,  
uint32_t mask ) [inline], [static]`

**23.7.12** `static void GPIO_PortToggle ( GPIO_Type * base, uint32_t port, uint32_t  
mask ) [inline], [static]`

### Parameters

<i>base</i>	GPIO peripheral base pointer(Typically GPIO)
<i>port</i>	GPIO port number
<i>mask</i>	GPIO pin number macro

**23.7.13** `static void GPIO_TogglePinsOutput ( GPIO_Type * base, uint32_t port,  
uint32_t mask ) [inline], [static]`

## Chapter 24

# GINT: Group GPIO Input Interrupt Driver

### 24.1 Overview

The MCUXpresso SDK provides a driver for the Group GPIO Input Interrupt (GINT).

It can configure one or more pins to generate a group interrupt when the pin conditions are met. The pins do not have to be configured as gpio pins.

### 24.2 Group GPIO Input Interrupt Driver operation

[GINT\\_SetCtrl\(\)](#) and [GINT\\_ConfigPins\(\)](#) functions configure the pins.

[GINT\\_EnableCallback\(\)](#) function enables the callback functionality. Callback function is called when the pin conditions are met.

### 24.3 Typical use case

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

#### Files

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

#### Typedefs

- typedef void(\* [gint\\_cb\\_t](#))(void)  
*GINT Callback function.*

#### Enumerations

- enum [gint\\_comb\\_t](#) {  
    [kGINT\\_CombineOr](#) = 0U,  
    [kGINT\\_CombineAnd](#) = 1U }  
*GINT combine inputs type.*
- enum [gint\\_trig\\_t](#) {  
    [kGINT\\_TrigEdge](#) = 0U,  
    [kGINT\\_TrigLevel](#) = 1U }  
*GINT trigger type.*

#### Functions

- void [GINT\\_Init](#) (GINT\_Type \*base)  
*Initialize GINT peripheral.*
- void [GINT\\_SetCtrl](#) (GINT\_Type \*base, [gint\\_comb\\_t](#) comb, [gint\\_trig\\_t](#) trig, [gint\\_cb\\_t](#) callback)

## Enumeration Type Documentation

- Setup GINT peripheral control parameters.*
  - void [GINT\\_GetCtrl](#) (GINT\_Type \*base, [gint\\_comb\\_t](#) \*comb, [gint\\_trig\\_t](#) \*trig, [gint\\_cb\\_t](#) \*callback)
- Get GINT peripheral control parameters.*
  - void [GINT\\_ConfigPins](#) (GINT\_Type \*base, [gint\\_port\\_t](#) port, uint32\_t polarityMask, uint32\_t enableMask)
- Configure GINT peripheral pins.*
  - void [GINT\\_GetConfigPins](#) (GINT\_Type \*base, [gint\\_port\\_t](#) port, uint32\_t \*polarityMask, uint32\_t \*enableMask)
- Get GINT peripheral pin configuration.*
  - void [GINT\\_EnableCallback](#) (GINT\_Type \*base)
- Enable callback.*
  - void [GINT\\_DisableCallback](#) (GINT\_Type \*base)
- Disable callback.*
  - static void [GINT\\_ClrStatus](#) (GINT\_Type \*base)
- Clear GINT status.*
  - static uint32\_t [GINT\\_GetStatus](#) (GINT\_Type \*base)
- Get GINT status.*
  - void [GINT\\_Deinit](#) (GINT\_Type \*base)
- Deinitialize GINT peripheral.*

## Driver version

- #define [FSL\\_GINT\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 0))  
Version 2.0.0.

## 24.4 Macro Definition Documentation

### 24.4.1 #define FSL\_GINT\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))

## 24.5 Typedef Documentation

### 24.5.1 typedef void(\* gint\_cb\_t)(void)

## 24.6 Enumeration Type Documentation

### 24.6.1 enum gint\_comb\_t

Enumerator

- kGINT\_CombineOr*** A grouped interrupt is generated when any one of the enabled inputs is active.
- kGINT\_CombineAnd*** A grouped interrupt is generated when all enabled inputs are active.

### 24.6.2 enum gint\_trig\_t

Enumerator

- kGINT\_TrigEdge*** Edge triggered based on polarity.
- kGINT\_TrigLevel*** Level triggered based on polarity.



## 24.7 Function Documentation

### 24.7.1 void GINT\_Init ( GINT\_Type \* *base* )

This function initializes the GINT peripheral and enables the clock.

## Function Documentation

### Parameters

<i>base</i>	Base address of the GINT peripheral.
-------------	--------------------------------------

### Return values

<i>None.</i>	
--------------	--

### 24.7.2 void GINT\_SetCtrl ( GINT\_Type \* *base*, gint\_comb\_t *comb*, gint\_trig\_t *trig*, gint\_cb\_t *callback* )

This function sets the control parameters of GINT peripheral.

### Parameters

<i>base</i>	Base address of the GINT peripheral.
<i>comb</i>	Controls if the enabled inputs are logically ORed or ANDed for interrupt generation.
<i>trig</i>	Controls if the enabled inputs are level or edge sensitive based on polarity.
<i>callback</i>	This function is called when configured group interrupt is generated.

### Return values

<i>None.</i>	
--------------	--

### 24.7.3 void GINT\_GetCtrl ( GINT\_Type \* *base*, gint\_comb\_t \* *comb*, gint\_trig\_t \* *trig*, gint\_cb\_t \* *callback* )

This function returns the control parameters of GINT peripheral.

### Parameters

<i>base</i>	Base address of the GINT peripheral.
<i>comb</i>	Pointer to store combine input value.
<i>trig</i>	Pointer to store trigger value.

<i>callback</i>	Pointer to store callback function.
-----------------	-------------------------------------

Return values

<i>None.</i>	
--------------	--

#### 24.7.4 void GINT\_ConfigPins ( GINT\_Type \* *base*, gint\_port\_t *port*, uint32\_t *polarityMask*, uint32\_t *enableMask* )

This function enables and controls the polarity of enabled pin(s) of a given port.

Parameters

<i>base</i>	Base address of the GINT peripheral.
<i>port</i>	Port number.
<i>polarityMask</i>	Each bit position selects the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
<i>enableMask</i>	Each bit position selects if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

<i>None.</i>	
--------------	--

#### 24.7.5 void GINT\_GetConfigPins ( GINT\_Type \* *base*, gint\_port\_t *port*, uint32\_t \* *polarityMask*, uint32\_t \* *enableMask* )

This function returns the pin configuration of a given port.

Parameters

<i>base</i>	Base address of the GINT peripheral.
<i>port</i>	Port number.
<i>polarityMask</i>	Pointer to store the polarity mask Each bit position indicates the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.

## Function Documentation

<i>enableMask</i>	Pointer to store the enable mask. Each bit position indicates if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.
-------------------	--

Return values

<i>None.</i>	
--------------	--

### 24.7.6 void GINT\_EnableCallback ( GINT\_Type \* *base* )

This function enables the interrupt for the selected GINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

<i>base</i>	Base address of the GINT peripheral.
-------------	--------------------------------------

Return values

<i>None.</i>	
--------------	--

### 24.7.7 void GINT\_DisableCallback ( GINT\_Type \* *base* )

This function disables the interrupt for the selected GINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

<i>base</i>	Base address of the peripheral.
-------------	---------------------------------

Return values

<i>None.</i>	
--------------	--

### 24.7.8 static void GINT\_ClrStatus ( GINT\_Type \* *base* ) [inline], [static]

This function clears the GINT status bit.

## Parameters

<i>base</i>	Base address of the GINT peripheral.
-------------	--------------------------------------

## Return values

<i>None.</i>	
--------------	--

### 24.7.9 static uint32\_t GINT\_GetStatus ( GINT\_Type \* *base* ) [inline], [static]

This function returns the GINT status.

## Parameters

<i>base</i>	Base address of the GINT peripheral.
-------------	--------------------------------------

## Return values

<i>status</i>	= 0 No group interrupt request. = 1 Group interrupt request active.
---------------	---

### 24.7.10 void GINT\_Deinit ( GINT\_Type \* *base* )

This function disables the GINT clock.

## Parameters

<i>base</i>	Base address of the GINT peripheral.
-------------	--------------------------------------

## Return values

<i>None.</i>	
--------------	--



## Chapter 25

# PINT: Pin Interrupt and Pattern Match Driver

### 25.1 Overview

The MCUXpresso SDK provides a driver for the Pin Interrupt and Pattern match (PINT).

It can configure one or more pins to generate a pin interrupt when the pin or pattern match conditions are met. The pins do not have to be configured as gpio pins however they must be connected to PINT via INPUTMUX. Only the pin interrupt or pattern match function can be active for interrupt generation. If the pin interrupt function is enabled then the pattern match function can be used for wakeup via RXEV.

### 25.2 Pin Interrupt and Pattern match Driver operation

[PINT\\_PinInterruptConfig\(\)](#) function configures the pins for pin interrupt.

[PINT\\_PatternMatchConfig\(\)](#) function configures the pins for pattern match.

#### 25.2.1 Pin Interrupt use case

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

#### 25.2.2 Pattern match use case

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

### Files

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

### Typedefs

- typedef void(\* [pint\\_cb\\_t](#))([pint\\_pin\\_int\\_t](#) pintr, uint32\_t pmatch\_status)  
*PINT Callback function.*

### Enumerations

- enum [pint\\_pin\\_enable\\_t](#) {  
    [kPINT\\_PinIntEnableNone](#) = 0U,  
    [kPINT\\_PinIntEnableRiseEdge](#) = PINT\_PIN\_RISE\_EDGE,  
    [kPINT\\_PinIntEnableFallEdge](#) = PINT\_PIN\_FALL\_EDGE,  
    [kPINT\\_PinIntEnableBothEdges](#) = PINT\_PIN\_BOTH\_EDGE,  
    [kPINT\\_PinIntEnableLowLevel](#) = PINT\_PIN\_LOW\_LEVEL,  
    [kPINT\\_PinIntEnableHighLevel](#) = PINT\_PIN\_HIGH\_LEVEL }

## Pin Interrupt and Pattern match Driver operation

- PINT Pin Interrupt enable type.*
  - enum `pint_pin_int_t` { `kPINT_PinInt0` = 0U }
- PINT Pin Interrupt type.*
  - enum `pint_pmatch_input_src_t` {  
`kPINT_PatternMatchInp0Src` = 0U,  
`kPINT_PatternMatchInp1Src` = 1U,  
`kPINT_PatternMatchInp2Src` = 2U,  
`kPINT_PatternMatchInp3Src` = 3U,  
`kPINT_PatternMatchInp4Src` = 4U,  
`kPINT_PatternMatchInp5Src` = 5U,  
`kPINT_PatternMatchInp6Src` = 6U,  
`kPINT_PatternMatchInp7Src` = 7U }
- PINT Pattern Match bit slice input source type.*
  - enum `pint_pmatch_bslic_t` { `kPINT_PatternMatchBSlice0` = 0U }
- PINT Pattern Match bit slice type.*
  - enum `pint_pmatch_bslic_cfg_t` {  
`kPINT_PatternMatchAlways` = 0U,  
`kPINT_PatternMatchStickyRise` = 1U,  
`kPINT_PatternMatchStickyFall` = 2U,  
`kPINT_PatternMatchStickyBothEdges` = 3U,  
`kPINT_PatternMatchHigh` = 4U,  
`kPINT_PatternMatchLow` = 5U,  
`kPINT_PatternMatchNever` = 6U,  
`kPINT_PatternMatchBothEdges` = 7U }
- PINT Pattern Match configuration type.*

## Functions

- void `PINT_Init` (`PINT_Type` \*base)  
*Initialize PINT peripheral.*
- void `PINT_PinInterruptConfig` (`PINT_Type` \*base, `pint_pin_int_t` intr, `pint_pin_enable_t` enable, `pint_cb_t` callback)  
*Configure PINT peripheral pin interrupt.*
- void `PINT_PinInterruptGetConfig` (`PINT_Type` \*base, `pint_pin_int_t` pintr, `pint_pin_enable_t` \*enable, `pint_cb_t` \*callback)  
*Get PINT peripheral pin interrupt configuration.*
- static void `PINT_PinInterruptClrStatus` (`PINT_Type` \*base, `pint_pin_int_t` pintr)  
*Clear Selected pin interrupt status.*
- static uint32\_t `PINT_PinInterruptGetStatus` (`PINT_Type` \*base, `pint_pin_int_t` pintr)  
*Get Selected pin interrupt status.*
- static void `PINT_PinInterruptClrStatusAll` (`PINT_Type` \*base)  
*Clear all pin interrupts status.*
- static uint32\_t `PINT_PinInterruptGetStatusAll` (`PINT_Type` \*base)  
*Get all pin interrupts status.*
- static void `PINT_PinInterruptClrFallFlag` (`PINT_Type` \*base, `pint_pin_int_t` pintr)  
*Clear Selected pin interrupt fall flag.*
- static uint32\_t `PINT_PinInterruptGetFallFlag` (`PINT_Type` \*base, `pint_pin_int_t` pintr)  
*Get selected pin interrupt fall flag.*
- static void `PINT_PinInterruptClrFallFlagAll` (`PINT_Type` \*base)



- *Clear all pin interrupt fall flags.*  
static uint32\_t [PINT\\_PinInterruptGetFallFlagAll](#) (PINT\_Type \*base)
- *Get all pin interrupt fall flags.*  
static void [PINT\\_PinInterruptClrRiseFlag](#) (PINT\_Type \*base, [pint\\_pin\\_int\\_t](#) pintr)
- *Clear Selected pin interrupt rise flag.*  
static uint32\_t [PINT\\_PinInterruptGetRiseFlag](#) (PINT\_Type \*base, [pint\\_pin\\_int\\_t](#) pintr)
- *Get selected pin interrupt rise flag.*  
static void [PINT\\_PinInterruptClrRiseFlagAll](#) (PINT\_Type \*base)
- *Clear all pin interrupt rise flags.*  
static uint32\_t [PINT\\_PinInterruptGetRiseFlagAll](#) (PINT\_Type \*base)
- *Get all pin interrupt rise flags.*  
void [PINT\\_PatternMatchConfig](#) (PINT\_Type \*base, [pint\\_pmatch\\_bslice\\_t](#) bslice, [pint\\_pmatch\\_cfg\\_t](#) \*cfg)
- *Configure PINT pattern match.*  
void [PINT\\_PatternMatchGetConfig](#) (PINT\_Type \*base, [pint\\_pmatch\\_bslice\\_t](#) bslice, [pint\\_pmatch\\_cfg\\_t](#) \*cfg)
- *Get PINT pattern match configuration.*  
static uint32\_t [PINT\\_PatternMatchGetStatus](#) (PINT\_Type \*base, [pint\\_pmatch\\_bslice\\_t](#) bslice)
- *Get pattern match bit slice status.*  
static uint32\_t [PINT\\_PatternMatchGetStatusAll](#) (PINT\_Type \*base)
- *Get status of all pattern match bit slices.*  
uint32\_t [PINT\\_PatternMatchResetDetectLogic](#) (PINT\_Type \*base)
- *Reset pattern match detection logic.*  
static void [PINT\\_PatternMatchEnable](#) (PINT\_Type \*base)
- *Enable pattern match function.*  
static void [PINT\\_PatternMatchDisable](#) (PINT\_Type \*base)
- *Disable pattern match function.*  
static void [PINT\\_PatternMatchEnableRXEV](#) (PINT\_Type \*base)
- *Enable RXEV output.*  
static void [PINT\\_PatternMatchDisableRXEV](#) (PINT\_Type \*base)
- *Disable RXEV output.*  
void [PINT\\_EnableCallback](#) (PINT\_Type \*base)
- *Enable callback.*  
void [PINT\\_DisableCallback](#) (PINT\_Type \*base)
- *Disable callback.*  
void [PINT\\_Deinit](#) (PINT\_Type \*base)
- *Deinitialize PINT peripheral.*

## Driver version

- #define [FSL\\_PINT\\_DRIVER\\_VERSION](#) ([MAKE\\_VERSION](#)(2, 0, 0))  
*Version 2.0.0.*

## 25.3 Typedef Documentation

### 25.3.1 typedef void(\* pint\_cb\_t)(pint\_pin\_int\_t pintr, uint32\_t pmatch\_status)

### 25.4 Enumeration Type Documentation

#### 25.4.1 enum pint\_pin\_enable\_t

Enumerator

- kPINT\_PinIntEnableNone* Do not generate Pin Interrupt.
- kPINT\_PinIntEnableRiseEdge* Generate Pin Interrupt on rising edge.
- kPINT\_PinIntEnableFallEdge* Generate Pin Interrupt on falling edge.
- kPINT\_PinIntEnableBothEdges* Generate Pin Interrupt on both edges.
- kPINT\_PinIntEnableLowLevel* Generate Pin Interrupt on low level.
- kPINT\_PinIntEnableHighLevel* Generate Pin Interrupt on high level.

#### 25.4.2 enum pint\_pin\_int\_t

Enumerator

- kPINT\_PinInt0* Pin Interrupt 0.

#### 25.4.3 enum pint\_pmatch\_input\_src\_t

Enumerator

- kPINT\_PatternMatchInp0Src* Input source 0.
- kPINT\_PatternMatchInp1Src* Input source 1.
- kPINT\_PatternMatchInp2Src* Input source 2.
- kPINT\_PatternMatchInp3Src* Input source 3.
- kPINT\_PatternMatchInp4Src* Input source 4.
- kPINT\_PatternMatchInp5Src* Input source 5.
- kPINT\_PatternMatchInp6Src* Input source 6.
- kPINT\_PatternMatchInp7Src* Input source 7.

#### 25.4.4 enum pint\_pmatch\_bslice\_t

Enumerator

- kPINT\_PatternMatchBSlice0* Bit slice 0.

### 25.4.5 enum pint\_pmatch\_bslice\_cfg\_t

Enumerator

*kPINT\_PatternMatchAlways* Always Contributes to product term match.  
*kPINT\_PatternMatchStickyRise* Sticky Rising edge.  
*kPINT\_PatternMatchStickyFall* Sticky Falling edge.  
*kPINT\_PatternMatchStickyBothEdges* Sticky Rising or Falling edge.  
*kPINT\_PatternMatchHigh* High level.  
*kPINT\_PatternMatchLow* Low level.  
*kPINT\_PatternMatchNever* Never contributes to product term match.  
*kPINT\_PatternMatchBothEdges* Either rising or falling edge.

## 25.5 Function Documentation

### 25.5.1 void PINT\_Init ( PINT\_Type \* *base* )

This function initializes the PINT peripheral and enables the clock.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>None.</i>	
--------------	--

### 25.5.2 void PINT\_PinInterruptConfig ( PINT\_Type \* *base*, pint\_pin\_int\_t *intr*, pint\_pin\_enable\_t *enable*, pint\_cb\_t *callback* )

This function configures a given pin interrupt.

Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>intr</i>	Pin interrupt.
<i>enable</i>	Selects detection logic.
<i>callback</i>	Callback.

## Function Documentation

Return values

<i>None.</i>	
--------------	--

### 25.5.3 void PINT\_PinInterruptGetConfig ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr*, pint\_pin\_enable\_t \* *enable*, pint\_cb\_t \* *callback* )

This function returns the configuration of a given pin interrupt.

Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.
<i>enable</i>	Pointer to store the detection logic.
<i>callback</i>	Callback.

Return values

<i>None.</i>	
--------------	--

### 25.5.4 static void PINT\_PinInterruptClrStatus ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr* ) [inline], [static]

This function clears the selected pin interrupt status.

Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.

Return values

<i>None.</i>	
--------------	--

### 25.5.5 static uint32\_t PINT\_PinInterruptGetStatus ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr* ) [inline], [static]

This function returns the selected pin interrupt status.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.

## Return values

<i>status</i>	= 0 No pin interrupt request. = 1 Selected Pin interrupt request active.
---------------	--

### 25.5.6 static void PINT\_PinInterruptClrStatusAll ( PINT\_Type \* *base* ) [inline], [static]

This function clears the status of all pin interrupts.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

## Return values

<i>None.</i>	
--------------	--

### 25.5.7 static uint32\_t PINT\_PinInterruptGetStatusAll ( PINT\_Type \* *base* ) [inline], [static]

This function returns the status of all pin interrupts.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

## Return values

<i>status</i>	Each bit position indicates the status of corresponding pin interrupt. = 0 No pin interrupt request. = 1 Pin interrupt request active.
---------------	--

### 25.5.8 static void PINT\_PinInterruptClrFallFlag ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr* ) [inline], [static]

This function clears the selected pin interrupt fall flag.

## Function Documentation

### Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.

### Return values

<i>None.</i>	
--------------	--

### 25.5.9 static uint32\_t PINT\_PinInterruptGetFallFlag ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr* ) [inline], [static]

This function returns the selected pin interrupt fall flag.

### Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.

### Return values

<i>flag</i>	= 0 Falling edge has not been detected. = 1 Falling edge has been detected.
-------------	---

### 25.5.10 static void PINT\_PinInterruptClrFallFlagAll ( PINT\_Type \* *base* ) [inline], [static]

This function clears the fall flag for all pin interrupts.

### Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

### Return values

<i>None.</i>	
--------------	--

### 25.5.11 static uint32\_t PINT\_PinInterruptGetFallFlagAll ( PINT\_Type \* *base* ) [inline], [static]

This function returns the fall flag of all pin interrupts.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

## Return values

<i>flags</i>	Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.
--------------	--

### 25.5.12 static void PINT\_PinInterruptClrRiseFlag ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr* ) [inline], [static]

This function clears the selected pin interrupt rise flag.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.

## Return values

<i>None.</i>	
--------------	--

### 25.5.13 static uint32\_t PINT\_PinInterruptGetRiseFlag ( PINT\_Type \* *base*, pint\_pin\_int\_t *pintr* ) [inline], [static]

This function returns the selected pin interrupt rise flag.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>pintr</i>	Pin interrupt.

## Return values

## Function Documentation

<i>flag</i>	= 0 Rising edge has not been detected. = 1 Rising edge has been detected.
-------------	---

### 25.5.14 static void PINT\_PinInterruptClrRiseFlagAll ( PINT\_Type \* *base* ) [inline], [static]

This function clears the rise flag for all pin interrupts.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>None.</i>	
--------------	--

### 25.5.15 static uint32\_t PINT\_PinInterruptGetRiseFlagAll ( PINT\_Type \* *base* ) [inline], [static]

This function returns the rise flag of all pin interrupts.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>flags</i>	Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.
--------------	---

### 25.5.16 void PINT\_PatternMatchConfig ( PINT\_Type \* *base*, pint\_pmatch\_bslice\_t *bslice*, pint\_pmatch\_cfg\_t \* *cfg* )

This function configures a given pattern match bit slice.



## Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>bslice</i>	Pattern match bit slice number.
<i>cfg</i>	Pointer to bit slice configuration.

## Return values

<i>None.</i>	
--------------	--

### 25.5.17 void PINT\_PatternMatchGetConfig ( PINT\_Type \* *base*, pint\_pmatch\_bslice\_t *bslice*, pint\_pmatch\_cfg\_t \* *cfg* )

This function returns the configuration of a given pattern match bit slice.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>bslice</i>	Pattern match bit slice number.
<i>cfg</i>	Pointer to bit slice configuration.

## Return values

<i>None.</i>	
--------------	--

### 25.5.18 static uint32\_t PINT\_PatternMatchGetStatus ( PINT\_Type \* *base*, pint\_pmatch\_bslice\_t *bslice* ) [inline], [static]

This function returns the status of selected bit slice.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
<i>bslice</i>	Pattern match bit slice number.

## Return values

---

## Function Documentation

<i>status</i>	= 0 Match has not been detected. = 1 Match has been detected.
---------------	---

### 25.5.19 static uint32\_t PINT\_PatternMatchGetStatusAll ( PINT\_Type \* *base* ) [inline], [static]

This function returns the status of all bit slices.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>status</i>	Each bit position indicates the match status of corresponding bit slice. = 0 Match has not been detected. = 1 Match has been detected.
---------------	--

### 25.5.20 uint32\_t PINT\_PatternMatchResetDetectLogic ( PINT\_Type \* *base* )

This function resets the pattern match detection logic if any of the product term is matching.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>pmstatus</i>	Each bit position indicates the match status of corresponding bit slice. = 0 Match was detected. = 1 Match was not detected.
-----------------	--

### 25.5.21 static void PINT\_PatternMatchEnable ( PINT\_Type \* *base* ) [inline], [static]

This function enables the pattern match function.

Parameters

---

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>None.</i>	
--------------	--

### 25.5.22 static void PINT\_PatternMatchDisable ( PINT\_Type \* *base* ) [inline], [static]

This function disables the pattern match function.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>None.</i>	
--------------	--

### 25.5.23 static void PINT\_PatternMatchEnableRXEV ( PINT\_Type \* *base* ) [inline], [static]

This function enables the pattern match RXEV output.

Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

Return values

<i>None.</i>	
--------------	--

### 25.5.24 static void PINT\_PatternMatchDisableRXEV ( PINT\_Type \* *base* ) [inline], [static]

This function disables the pattern match RXEV output.

## Function Documentation

### Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

### Return values

<i>None.</i>	
--------------	--

### 25.5.25 void PINT\_EnableCallback ( PINT\_Type \* *base* )

This function enables the interrupt for the selected PINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

### Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

### Return values

<i>None.</i>	
--------------	--

### 25.5.26 void PINT\_DisableCallback ( PINT\_Type \* *base* )

This function disables the interrupt for the selected PINT peripheral. Although the pins are still being monitored but the callback function is not called.

### Parameters

<i>base</i>	Base address of the peripheral.
-------------	---------------------------------

### Return values

<i>None.</i>	
--------------	--

### 25.5.27 void PINT\_Deinit ( PINT\_Type \* *base* )

This function disables the PINT clock.

## Parameters

<i>base</i>	Base address of the PINT peripheral.
-------------	--------------------------------------

## Return values

<i>None.</i>	
--------------	--



## Chapter 26

### CTIMER: Standard counter/timers

#### 26.1 Overview

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

#### 26.2 Function groups

The ctimer driver supports the generation of PWM signals, input capture and setting up the timer match conditions.

##### 26.2.1 Initialization and deinitialization

The function [CTIMER\\_Init\(\)](#) initializes the ctimer with specified configurations. The function [CTIMER\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the counter/timer mode and input selection when running in counter mode.

The function [CTIMER\\_Deinit\(\)](#) stops the timer and turns off the module clock.

##### 26.2.2 PWM Operations

The function [CTIMER\\_SetupPwm\(\)](#) sets up channels for PWM output. Each channel has its own duty cycle, however the same PWM period 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 [CTIMER\\_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular channel.

##### 26.2.3 Match Operation

The function [CTIMER\\_SetupMatch\(\)](#) sets up channels for match operation. Each channel is configured with a match value, if the counter should stop on match, if counter should reset on match and output pin action. The output signal can be cleared, set or toggled on match.

##### 26.2.4 Input capture operations

The function [CTIMER\\_SetupCapture\(\)](#) sets up an channel for input capture. The user can specify the capture edge and if a interrupt should be generated when processing the input signal.

## Typical use case

### 26.3 Typical use case

#### 26.3.1 Match example

Set up a match channel to toggle output when a match occurs. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ctimer

#### 26.3.2 PWM output example

Set up a channel for PWM output. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ctimer

## Files

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

## Data Structures

- struct [ctimer\\_match\\_config\\_t](#)  
*Match configuration. [More...](#)*
- struct [ctimer\\_config\\_t](#)  
*Timer configuration structure. [More...](#)*

## Enumerations

- enum [ctimer\\_capture\\_channel\\_t](#) {  
    [kCTIMER\\_Capture\\_0](#) = 0U,  
    [kCTIMER\\_Capture\\_1](#),  
    [kCTIMER\\_Capture\\_2](#) }  
*List of Timer capture channels.*
- enum [ctimer\\_capture\\_edge\\_t](#) {  
    [kCTIMER\\_Capture\\_RiseEdge](#) = 1U,  
    [kCTIMER\\_Capture\\_FallEdge](#) = 2U,  
    [kCTIMER\\_Capture\\_BothEdge](#) = 3U }  
*List of capture edge options.*
- enum [ctimer\\_match\\_t](#) {  
    [kCTIMER\\_Match\\_0](#) = 0U,  
    [kCTIMER\\_Match\\_1](#),  
    [kCTIMER\\_Match\\_2](#),  
    [kCTIMER\\_Match\\_3](#) }  
*List of Timer match registers.*
- enum [ctimer\\_match\\_output\\_control\\_t](#) {  
    [kCTIMER\\_Output\\_NoAction](#) = 0U,  
    [kCTIMER\\_Output\\_Clear](#),  
    [kCTIMER\\_Output\\_Set](#),  
    [kCTIMER\\_Output\\_Toggle](#) }



- *List of output control options.*
- enum `ctimer_timer_mode_t`
- *List of Timer modes.*
- enum `ctimer_interrupt_enable_t` {  
`kCTIMER_Match0InterruptEnable` = `CTIMER_MCR_MR0I_MASK`,  
`kCTIMER_Match1InterruptEnable` = `CTIMER_MCR_MR1I_MASK`,  
`kCTIMER_Match2InterruptEnable` = `CTIMER_MCR_MR2I_MASK`,  
`kCTIMER_Match3InterruptEnable` = `CTIMER_MCR_MR3I_MASK`,  
`kCTIMER_Capture0InterruptEnable` = `CTIMER_CCR_CAP0I_MASK`,  
`kCTIMER_Capture1InterruptEnable` = `CTIMER_CCR_CAP1I_MASK`,  
`kCTIMER_Capture2InterruptEnable` = `CTIMER_CCR_CAP2I_MASK` }
- *List of Timer interrupts.*
- enum `ctimer_status_flags_t` {  
`kCTIMER_Match0Flag` = `CTIMER_IR_MR0INT_MASK`,  
`kCTIMER_Match1Flag` = `CTIMER_IR_MR1INT_MASK`,  
`kCTIMER_Match2Flag` = `CTIMER_IR_MR2INT_MASK`,  
`kCTIMER_Match3Flag` = `CTIMER_IR_MR3INT_MASK`,  
`kCTIMER_Capture0Flag` = `CTIMER_IR_CR0INT_MASK`,  
`kCTIMER_Capture1Flag` = `CTIMER_IR_CR1INT_MASK`,  
`kCTIMER_Capture2Flag` = `CTIMER_IR_CR2INT_MASK` }
- *List of Timer flags.*
- enum `ctimer_callback_type_t` {  
`kCTIMER_SingleCallback`,  
`kCTIMER_MultipleCallback` }
- *Callback type when registering for a callback.*

## Functions

- void `CTIMER_SetupMatch` (`CTIMER_Type *base`, `ctimer_match_t matchChannel`, const `ctimer_match_config_t *config`)  
*Setup the match register.*
- void `CTIMER_SetupCapture` (`CTIMER_Type *base`, `ctimer_capture_channel_t capture`, `ctimer_capture_edge_t edge`, bool `enableInt`)  
*Setup the capture.*
- void `CTIMER_RegisterCallBack` (`CTIMER_Type *base`, `ctimer_callback_t *cb_func`, `ctimer_callback_type_t cb_type`)  
*Register callback.*
- static void `CTIMER_Reset` (`CTIMER_Type *base`)  
*Reset the counter.*

## Driver version

- #define `FSL_CTIMER_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*Version 2.0.0.*

## Initialization and deinitialization

- void `CTIMER_Init` (`CTIMER_Type *base`, const `ctimer_config_t *config`)

## Data Structure Documentation

- *Ungates the clock and configures the peripheral for basic operation.*  
void [CTIMER\\_Deinit](#) (CTIMER\_Type \*base)
- *Gates the timer clock.*  
void [CTIMER\\_GetDefaultConfig](#) (ctimer\_config\_t \*config)  
*Fills in the timers configuration structure with the default settings.*

## PWM setup operations

- [status\\_t CTIMER\\_SetupPwm](#) (CTIMER\_Type \*base, [ctimer\\_match\\_t](#) matchChannel, uint8\_t dutyCyclePercent, uint32\_t pwmFreq\_Hz, uint32\_t srcClock\_Hz, bool enableInt)  
*Configures the PWM signal parameters.*
- void [CTIMER\\_UpdatePwmDutycycle](#) (CTIMER\_Type \*base, [ctimer\\_match\\_t](#) matchChannel, uint8\_t dutyCyclePercent)  
*Updates the duty cycle of an active PWM signal.*

## Interrupt Interface

- static void [CTIMER\\_EnableInterrupts](#) (CTIMER\_Type \*base, uint32\_t mask)  
*Enables the selected Timer interrupts.*
- static void [CTIMER\\_DisableInterrupts](#) (CTIMER\_Type \*base, uint32\_t mask)  
*Disables the selected Timer interrupts.*
- static uint32\_t [CTIMER\\_GetEnabledInterrupts](#) (CTIMER\_Type \*base)  
*Gets the enabled Timer interrupts.*

## Status Interface

- static uint32\_t [CTIMER\\_GetStatusFlags](#) (CTIMER\_Type \*base)  
*Gets the Timer status flags.*
- static void [CTIMER\\_ClearStatusFlags](#) (CTIMER\_Type \*base, uint32\_t mask)  
*Clears the Timer status flags.*

## Counter Start and Stop

- static void [CTIMER\\_StartTimer](#) (CTIMER\_Type \*base)  
*Starts the Timer counter.*
- static void [CTIMER\\_StopTimer](#) (CTIMER\_Type \*base)  
*Stops the Timer counter.*

## 26.4 Data Structure Documentation

### 26.4.1 struct ctimer\_match\_config\_t

This structure holds the configuration settings for each match register.

#### Data Fields

- uint32\_t [matchValue](#)  
*This is stored in the match register.*

- bool [enableCounterReset](#)  
*true: Match will reset the counter false: Match will not reset the counter*
- bool [enableCounterStop](#)  
*true: Match will stop the counter false: Match will not stop the counter*
- [ctimer\\_match\\_output\\_control\\_t](#) [outControl](#)  
*Action to be taken on a match on the EM bit/output.*
- bool [outPinInitState](#)  
*Initial value of the EM bit/output.*
- bool [enableInterrupt](#)  
*true: Generate interrupt upon match false: Do not generate interrupt on match*

### 26.4.2 struct ctimer\_config\_t

This structure holds the configuration settings for the Timer peripheral. To initialize this structure to reasonable defaults, call the [CTIMER\\_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

- [ctimer\\_timer\\_mode\\_t](#) [mode](#)  
*Timer mode.*
- [ctimer\\_capture\\_channel\\_t](#) [input](#)  
*Input channel to increment the timer, used only in timer modes that rely on this input signal to increment TC.*
- uint32\_t [prescale](#)  
*Prescale value.*

## 26.5 Enumeration Type Documentation

### 26.5.1 enum ctimer\_capture\_channel\_t

Enumerator

***kCTIMER\_Capture\_0*** Timer capture channel 0.  
***kCTIMER\_Capture\_1*** Timer capture channel 1.  
***kCTIMER\_Capture\_2*** Timer capture channel 2.

### 26.5.2 enum ctimer\_capture\_edge\_t

Enumerator

***kCTIMER\_Capture\_RiseEdge*** Capture on rising edge.  
***kCTIMER\_Capture\_FallEdge*** Capture on falling edge.

## Enumeration Type Documentation

***kCTIMER\_Capture\_BothEdge*** Capture on rising and falling edge.

### 26.5.3 enum ctimer\_match\_t

Enumerator

***kCTIMER\_Match\_0*** Timer match register 0.

***kCTIMER\_Match\_1*** Timer match register 1.

***kCTIMER\_Match\_2*** Timer match register 2.

***kCTIMER\_Match\_3*** Timer match register 3.

### 26.5.4 enum ctimer\_match\_output\_control\_t

Enumerator

***kCTIMER\_Output\_NoAction*** No action is taken.

***kCTIMER\_Output\_Clear*** Clear the EM bit/output to 0.

***kCTIMER\_Output\_Set*** Set the EM bit/output to 1.

***kCTIMER\_Output\_Toggle*** Toggle the EM bit/output.

### 26.5.5 enum ctimer\_interrupt\_enable\_t

Enumerator

***kCTIMER\_Match0InterruptEnable*** Match 0 interrupt.

***kCTIMER\_Match1InterruptEnable*** Match 1 interrupt.

***kCTIMER\_Match2InterruptEnable*** Match 2 interrupt.

***kCTIMER\_Match3InterruptEnable*** Match 3 interrupt.

***kCTIMER\_Capture0InterruptEnable*** Capture 0 interrupt.

***kCTIMER\_Capture1InterruptEnable*** Capture 1 interrupt.

***kCTIMER\_Capture2InterruptEnable*** Capture 2 interrupt.

### 26.5.6 enum ctimer\_status\_flags\_t

Enumerator

***kCTIMER\_Match0Flag*** Match 0 interrupt flag.

***kCTIMER\_Match1Flag*** Match 1 interrupt flag.

***kCTIMER\_Match2Flag*** Match 2 interrupt flag.

***kCTIMER\_Match3Flag*** Match 3 interrupt flag.

***kCTIMER\_Capture0Flag*** Capture 0 interrupt flag.

***kCTIMER\_Capture1Flag*** Capture 1 interrupt flag.

***kCTIMER\_Capture2Flag*** Capture 2 interrupt flag.

### 26.5.7 enum ctimer\_callback\_type\_t

When registering a callback an array of function pointers is passed the size could be 1 or 8, the callback type will tell that.

Enumerator

***kCTIMER\_SingleCallback*** Single Callback type where there is only one callback for the timer. based on the status flags different channels needs to be handled differently

***kCTIMER\_MultipleCallback*** Multiple Callback type where there can be 8 valid callbacks, one per channel. for both match/capture

## 26.6 Function Documentation

### 26.6.1 void CTIMER\_Init ( CTIMER\_Type \* *base*, const ctimer\_config\_t \* *config* )

Note

This API should be called at the beginning of the application before using the driver.

Parameters

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

### 26.6.2 void CTIMER\_Deinit ( CTIMER\_Type \* *base* )

Parameters

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

### 26.6.3 void CTIMER\_GetDefaultConfig ( ctimer\_config\_t \* *config* )

The default values are:

```
* config->mode = kCTIMER_TimerMode;
* config->input = kCTIMER_Capture_0;
* config->prescale = 0;
*
```

## Function Documentation

### Parameters

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

#### 26.6.4 **status\_t CTIMER\_SetupPwm ( CTIMER\_Type \* *base*, ctimer\_match\_t *matchChannel*, uint8\_t *dutyCyclePercent*, uint32\_t *pwmFreq\_Hz*, uint32\_t *srcClock\_Hz*, bool *enableInt* )**

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function will assign match channel 3 to set the PWM cycle.

### Note

When setting PWM output from multiple output pins, all should use the same PWM frequency

### Parameters

<i>base</i>	Ctimer peripheral base address
<i>matchChannel</i>	Match pin to be used to output the PWM signal
<i>dutyCycle-Percent</i>	PWM pulse width; the value should be between 0 to 100
<i>pwmFreq_Hz</i>	PWM signal frequency in Hz
<i>srcClock_Hz</i>	Timer counter clock in Hz
<i>enableInt</i>	Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt is generated

### Returns

kStatus\_Success on success kStatus\_Fail If matchChannel passed in is 3; this channel is reserved to set the PWM cycle

#### 26.6.5 **void CTIMER\_UpdatePwmDutycycle ( CTIMER\_Type \* *base*, ctimer\_match\_t *matchChannel*, uint8\_t *dutyCyclePercent* )**

## Parameters

<i>base</i>	Ctimer peripheral base address
<i>matchChannel</i>	Match pin to be used to output the PWM signal
<i>dutyCycle-Percent</i>	New PWM pulse width; the value should be between 0 to 100

### 26.6.6 void CTIMER\_SetupMatch ( CTIMER\_Type \* *base*, ctimer\_match\_t *matchChannel*, const ctimer\_match\_config\_t \* *config* )

User configuration is used to setup the match value and action to be taken when a match occurs.

## Parameters

<i>base</i>	Ctimer peripheral base address
<i>matchChannel</i>	Match register to configure
<i>config</i>	Pointer to the match configuration structure

### 26.6.7 void CTIMER\_SetupCapture ( CTIMER\_Type \* *base*, ctimer\_capture\_channel\_t *capture*, ctimer\_capture\_edge\_t *edge*, bool *enableInt* )

## Parameters

<i>base</i>	Ctimer peripheral base address
<i>capture</i>	Capture channel to configure
<i>edge</i>	Edge on the channel that will trigger a capture
<i>enableInt</i>	Flag to enable channel interrupts, if enabled then the registered call back is called upon capture

### 26.6.8 void CTIMER\_RegisterCallBack ( CTIMER\_Type \* *base*, ctimer\_callback\_t \* *cb\_func*, ctimer\_callback\_type\_t *cb\_type* )

## Function Documentation

### Parameters

<i>base</i>	Ctimer peripheral base address
<i>cb_func</i>	callback function
<i>cb_type</i>	callback function type, singular or multiple

### 26.6.9 static void CTIMER\_EnableInterrupts ( CTIMER\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

### Parameters

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

### 26.6.10 static void CTIMER\_DisableInterrupts ( CTIMER\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

### Parameters

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

### 26.6.11 static uint32\_t CTIMER\_GetEnabledInterrupts ( CTIMER\_Type \* *base* ) [inline], [static]

### Parameters

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

### Returns

The enabled interrupts. This is the logical OR of members of the enumeration [ctimer\\_interrupt\\_enable\\_t](#)



**26.6.12** `static uint32_t CTIMER_GetStatusFlags ( CTIMER_Type * base )`  
`[inline], [static]`

## Function Documentation

### Parameters

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

### Returns

The status flags. This is the logical OR of members of the enumeration [ctimer\\_status\\_flags\\_t](#)

**26.6.13 static void CTIMER\_ClearStatusFlags ( CTIMER\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

### Parameters

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

**26.6.14 static void CTIMER\_StartTimer ( CTIMER\_Type \* *base* ) [inline], [static]**

### Parameters

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

**26.6.15 static void CTIMER\_StopTimer ( CTIMER\_Type \* *base* ) [inline], [static]**

### Parameters

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

**26.6.16 static void CTIMER\_Reset ( CTIMER\_Type \* *base* ) [inline], [static]**

The timer counter and prescale counter are reset on the next positive edge of the APB clock.

Parameters

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



## Chapter 27

### Common Driver

#### 27.1 Overview

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

#### Macros

- #define **MAKE\_STATUS**(group, code) (((group)\*100) + (code))  
*Construct a status code value from a group and code number.*
- #define **MAKE\_VERSION**(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))  
*Construct the version number for drivers.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_NONE** 0U  
*No debug console.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_UART** 1U  
*Debug console base on UART.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_LPUART** 2U  
*Debug console base on LPUART.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_LPSCI** 3U  
*Debug console base on LPSCI.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_USBCDC** 4U  
*Debug console base on USBCDC.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_FLEXCOMM** 5U  
*Debug console base on USBCDC.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_IUART** 6U  
*Debug console base on i.MX UART.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_VUSART** 7U  
*Debug console base on LPC\_USART.*
- #define **DEBUG\_CONSOLE\_DEVICE\_TYPE\_MINI\_USART** 8U  
*Debug console base on LPC\_USART.*
- #define **ARRAY\_SIZE**(x) (sizeof(x) / sizeof((x)[0]))  
*Computes the number of elements in an array.*

#### Typedefs

- typedef int32\_t **status\_t**  
*Type used for all status and error return values.*

### Enumerations

- enum \_status\_groups {
  - kStatusGroup\_Generic = 0,
  - kStatusGroup\_FLASH = 1,
  - kStatusGroup\_LPSPI = 4,
  - kStatusGroup\_FLEXIO\_SPI = 5,
  - kStatusGroup\_DSPI = 6,
  - kStatusGroup\_FLEXIO\_UART = 7,
  - kStatusGroup\_FLEXIO\_I2C = 8,
  - kStatusGroup\_LPI2C = 9,
  - kStatusGroup\_UART = 10,
  - kStatusGroup\_I2C = 11,
  - kStatusGroup\_LPSCI = 12,
  - kStatusGroup\_LPUART = 13,
  - kStatusGroup\_SPI = 14,
  - kStatusGroup\_XRDC = 15,
  - kStatusGroup\_SEMA42 = 16,
  - kStatusGroup\_SDHC = 17,
  - kStatusGroup\_SDMMC = 18,
  - kStatusGroup\_SAI = 19,
  - kStatusGroup\_MCG = 20,
  - kStatusGroup\_SCG = 21,
  - kStatusGroup\_SDSPI = 22,
  - kStatusGroup\_FLEXIO\_I2S = 23,
  - kStatusGroup\_FLEXIO\_MCULCD = 24,
  - kStatusGroup\_FLASHIAP = 25,
  - kStatusGroup\_FLEXCOMM\_I2C = 26,
  - kStatusGroup\_I2S = 27,
  - kStatusGroup\_IUART = 28,
  - kStatusGroup\_CSI = 29,
  - kStatusGroup\_MIPI\_DSI = 30,
  - kStatusGroup\_SDRAMC = 35,
  - kStatusGroup\_POWER = 39,
  - kStatusGroup\_ENET = 40,
  - kStatusGroup\_PHY = 41,
  - kStatusGroup\_TRGMUX = 42,
  - kStatusGroup\_SMARTCARD = 43,
  - kStatusGroup\_LMEM = 44,
  - kStatusGroup\_QSPI = 45,
  - kStatusGroup\_DMA = 50,
  - kStatusGroup\_EDMA = 51,
  - kStatusGroup\_DMAMGR = 52,
  - kStatusGroup\_FLEXCAN = 53,
  - kStatusGroup\_LTC = 54,
  - kStatusGroup\_FLEXIO\_CAMERA = 55,
  - kStatusGroup\_LPC\_SPI = 56,
  - kStatusGroup\_LPC\_USMCA = 57,
  - kStatusGroup\_DMIC = 58,
  - kStatusGroup\_SDIF = 59,

```
kStatusGroup_ApplicationRangeStart = 101 }
```

*Status group numbers.*

- enum `_generic_status`

*Generic status return codes.*

## Functions

- static `status_t EnableIRQ` (IRQn\_Type interrupt)  
*Enable specific interrupt.*
- static `status_t DisableIRQ` (IRQn\_Type interrupt)  
*Disable specific interrupt.*
- static `uint32_t DisableGlobalIRQ` (void)  
*Disable the global IRQ.*
- static void `EnableGlobalIRQ` (uint32\_t primask)  
*Enable the global IRQ.*
- void `EnableDeepSleepIRQ` (IRQn\_Type interrupt)  
*Enable specific interrupt for wake-up from deep-sleep mode.*
- void `DisableDeepSleepIRQ` (IRQn\_Type interrupt)  
*Disable specific interrupt for wake-up from deep-sleep mode.*

## Driver version

- #define `FSL_COMMON_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 0))  
*common driver version 2.0.0.*

## Min/max macros

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

## UINT16\_MAX/UINT32\_MAX value

- #define `UINT16_MAX` ((uint16\_t)-1)
- #define `UINT32_MAX` ((uint32\_t)-1)

## Timer utilities

- #define `USEC_TO_COUNT`(us, clockFreqInHz) (uint64\_t)((uint64\_t)us \* clockFreqInHz / 1000000U)  
*Macro to convert a microsecond period to raw count value.*
- #define `COUNT_TO_USEC`(count, clockFreqInHz) (uint64\_t)((uint64\_t)count \* 1000000U / clockFreqInHz)  
*Macro to convert a raw count value to microsecond.*
- #define `MSEC_TO_COUNT`(ms, clockFreqInHz) (uint64\_t)((uint64\_t)ms \* clockFreqInHz / 1000U)  
*Macro to convert a millisecond period to raw count value.*
- #define `COUNT_TO_MSEC`(count, clockFreqInHz) (uint64\_t)((uint64\_t)count \* 1000U / clockFreqInHz)  
*Macro to convert a raw count value to millisecond.*

### Alignment variable definition macros

- #define **SDK\_ALIGN**(var, alignbytes) var
  - #define **SDK\_SIZEALIGN**(var, alignbytes) (((unsigned int)((var) + ((alignbytes)-1)) & (unsigned int)(~(unsigned int)((alignbytes)-1)))
- Macro to change a value to a given size aligned value.*

### Non-cacheable region definition macros

- #define **AT\_NONCACHEABLE\_SECTION**(var) var
- #define **AT\_NONCACHEABLE\_SECTION\_ALIGN**(var, alignbytes) var
- #define **AT\_NONCACHEABLE\_SECTION\_INIT**(var) var
- #define **AT\_NONCACHEABLE\_SECTION\_ALIGN\_INIT**(var, alignbytes) var

## 27.2 Macro Definition Documentation

**27.2.1 #define MAKE\_STATUS( group, code ) (((group)\*100) + (code)))**

**27.2.2 #define MAKE\_VERSION( major, minor, bugfix ) (((major) << 16) | ((minor) << 8) | (bugfix))**

**27.2.3 #define FSL\_COMMON\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 0))**

**27.2.4 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_NONE 0U**

**27.2.5 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_UART 1U**

**27.2.6 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_LPUART 2U**

**27.2.7 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_LPSCI 3U**

**27.2.8 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_USBCDC 4U**

**27.2.9 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_FLEXCOMM 5U**

**27.2.10 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_IUART 6U**

**27.2.11 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_VUSART 7U**

**27.2.12 #define DEBUG\_CONSOLE\_DEVICE\_TYPE\_MINI\_USART 8U**

**27.2.13 #define ARRAY\_SIZE( x ) (sizeof(x) / sizeof((x)[0]))**



## 27.3 Typedef Documentation

### 27.3.1 typedef int32\_t status\_t

## 27.4 Enumeration Type Documentation

### 27.4.1 enum \_status\_groups

Enumerator

*kStatusGroup\_Generic* Group number for generic status codes.  
*kStatusGroup\_FLASH* Group number for FLASH status codes.  
*kStatusGroup\_LPSPI* Group number for LPSPI status codes.  
*kStatusGroup\_FLEXIO\_SPI* Group number for FLEXIO SPI status codes.  
*kStatusGroup\_DSPI* Group number for DSPI status codes.  
*kStatusGroup\_FLEXIO\_UART* Group number for FLEXIO UART status codes.  
*kStatusGroup\_FLEXIO\_I2C* Group number for FLEXIO I2C status codes.  
*kStatusGroup\_LPI2C* Group number for LPI2C status codes.  
*kStatusGroup\_UART* Group number for UART status codes.  
*kStatusGroup\_I2C* Group number for UART status codes.  
*kStatusGroup\_LPSCI* Group number for LPSCI status codes.  
*kStatusGroup\_LPUART* Group number for LPUART status codes.  
*kStatusGroup\_SPI* Group number for SPI status code.  
*kStatusGroup\_XRDC* Group number for XRDC status code.  
*kStatusGroup\_SEMA42* Group number for SEMA42 status code.  
*kStatusGroup\_SDHC* Group number for SDHC status code.  
*kStatusGroup\_SDMMC* Group number for SDMMC status code.  
*kStatusGroup\_SAI* Group number for SAI status code.  
*kStatusGroup\_MCG* Group number for MCG status codes.  
*kStatusGroup\_SCG* Group number for SCG status codes.  
*kStatusGroup\_SDSPI* Group number for SDSPI status codes.  
*kStatusGroup\_FLEXIO\_I2S* Group number for FLEXIO I2S status codes.  
*kStatusGroup\_FLEXIO\_MCULCD* Group number for FLEXIO LCD status codes.  
*kStatusGroup\_FLASHIAP* Group number for FLASHIAP status codes.  
*kStatusGroup\_FLEXCOMM\_I2C* Group number for FLEXCOMM I2C status codes.  
*kStatusGroup\_I2S* Group number for I2S status codes.  
*kStatusGroup\_IUART* Group number for IUART status codes.  
*kStatusGroup\_CSI* Group number for CSI status codes.  
*kStatusGroup\_MIPI\_DSI* Group number for MIPI DSI status codes.  
*kStatusGroup\_SDRAMC* Group number for SDRAMC status codes.  
*kStatusGroup\_POWER* Group number for POWER status codes.  
*kStatusGroup\_ENET* Group number for ENET status codes.  
*kStatusGroup\_PHY* Group number for PHY status codes.  
*kStatusGroup\_TRGMUX* Group number for TRGMUX status codes.  
*kStatusGroup\_SMARTCARD* Group number for SMARTCARD status codes.  
*kStatusGroup\_LMEM* Group number for LMEM status codes.

## Function Documentation

*kStatusGroup\_QSPI* Group number for QSPI status codes.  
*kStatusGroup\_DMA* Group number for DMA status codes.  
*kStatusGroup\_EDMA* Group number for EDMA status codes.  
*kStatusGroup\_DMAMGR* Group number for DMAMGR status codes.  
*kStatusGroup\_FLEXCAN* Group number for FlexCAN status codes.  
*kStatusGroup\_LTC* Group number for LTC status codes.  
*kStatusGroup\_FLEXIO\_CAMERA* Group number for FLEXIO CAMERA status codes.  
*kStatusGroup\_LPC\_SPI* Group number for LPC\_SPI status codes.  
*kStatusGroup\_LPC\_USART* Group number for LPC\_USART status codes.  
*kStatusGroup\_DMIC* Group number for DMIC status codes.  
*kStatusGroup\_SDIF* Group number for SDIF status codes.  
*kStatusGroup\_SPIFI* Group number for SPIFI status codes.  
*kStatusGroup\_OTP* Group number for OTP status codes.  
*kStatusGroup\_MCAN* Group number for MCAN status codes.  
*kStatusGroup\_CAAM* Group number for CAAM status codes.  
*kStatusGroup\_ECSPi* Group number for ECSPi status codes.  
*kStatusGroup\_USDHC* Group number for USDHC status codes.  
*kStatusGroup\_LPC\_I2C* Group number for LPC\_I2C status codes.  
*kStatusGroup\_DCP* Group number for DCP status codes.  
*kStatusGroup\_MSCAN* Group number for MSCAN status codes.  
*kStatusGroup\_ESAI* Group number for ESAI status codes.  
*kStatusGroup\_FLEXSPI* Group number for FLEXSPI status codes.  
*kStatusGroup\_MMDC* Group number for MMDC status codes.  
*kStatusGroup\_MICFIL* Group number for MIC status codes.  
*kStatusGroup\_SDMA* Group number for SDMA status codes.  
*kStatusGroup\_ICS* Group number for ICS status codes.  
*kStatusGroup\_SPDIF* Group number for SPDIF status codes.  
*kStatusGroup\_NOTIFIER* Group number for NOTIFIER status codes.  
*kStatusGroup\_DebugConsole* Group number for debug console status codes.  
*kStatusGroup\_SEMC* Group number for SEMC status codes.  
*kStatusGroup\_ApplicationRangeStart* Starting number for application groups.

### 27.4.2 enum \_generic\_status

## 27.5 Function Documentation

### 27.5.1 static status\_t EnableIRQ ( IRQn\_Type *interrupt* ) [inline], [static]

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

## Function Documentation

### Parameters

<i>interrupt</i>	The IRQ number.
------------------	-----------------

### Return values

<i>kStatus_Success</i>	Interrupt enabled successfully
<i>kStatus_Fail</i>	Failed to enable the interrupt

### 27.5.2 static status\_t DisableIRQ ( IRQn\_Type *interrupt* ) [inline], [static]

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro FSL\_FEATURE\_NUMBER\_OF\_LEVEL1\_INT\_VECTORS.

### Parameters

<i>interrupt</i>	The IRQ number.
------------------	-----------------

### Return values

<i>kStatus_Success</i>	Interrupt disabled successfully
<i>kStatus_Fail</i>	Failed to disable the interrupt

### 27.5.3 static uint32\_t DisableGlobalIRQ ( void ) [inline], [static]

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the [EnableGlobalIRQ\(\)](#).

### Returns

Current primask value.

### 27.5.4 static void EnableGlobalIRQ ( uint32\_t *primask* ) [inline], [static]

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convinience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the [EnableGlobalIRQ\(\)](#) and [DisableGlobalIRQ\(\)](#) in pair.

## Parameters

<i>primask</i>	value of primask register to be restored. The primask value is supposed to be provided by the <a href="#">DisableGlobalIRQ()</a> .
----------------	--

**27.5.5 void EnableDeepSleepIRQ ( IRQn\_Type *interrupt* )**

Enable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

## Note

This function also enables the interrupt in the NVIC ([EnableIRQ\(\)](#) is called internally).

## Parameters

<i>interrupt</i>	The IRQ number.
------------------	-----------------

**27.5.6 void DisableDeepSleepIRQ ( IRQn\_Type *interrupt* )**

Disable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

## Note

This function also disables the interrupt in the NVIC ([DisableIRQ\(\)](#) is called internally).

## Parameters

<i>interrupt</i>	The IRQ number.
------------------	-----------------



## Chapter 28 Debug Console

### 28.1 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. It consists of log, str, io. Log layer is used to handle the formatted log, push log to buffer or flush log to IO. STR layer is used to format the printf and scanf log. IO layer is a wrapper of various uart peripheral.

### 28.2 Function groups

#### 28.2.1 Initialization

To initialize the debug console, call the [DbgConsole\\_Init\(\)](#) function with these parameters. This function automatically enables the module and the clock.

```
/*
 * @brief Initializes the peripheral used to debug messages.
 *
 * @param baseAddr      Indicates which address of the peripheral is used to send debug messages.
 * @param baudRate      The desired baud rate in bits per second.
 * @param device        Low level device type for the debug console, can be one of:
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_UART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPUART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPSCI,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_USBCDC.
 * @param clkSrcFreq    Frequency of peripheral source clock.
 *
 * @return              Whether initialization was successful or not.
 */
status_t DbgConsole_Init(uint32_t baseAddr, uint32_t baudRate, uint8_t device,
                        uint32_t clkSrcFreq)
```

Selects the supported debug console hardware device type, such as

```
DEBUG_CONSOLE_DEVICE_TYPE_NONE
DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
DEBUG_CONSOLE_DEVICE_TYPE_UART
DEBUG_CONSOLE_DEVICE_TYPE_LPUART
DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral.

This example shows how to call the [DbgConsole\\_Init\(\)](#) given the user configuration structure.

```
uint32_t uartClkSrcFreq = CLOCK_GetFreq(BOARD_DEBUG_UART_CLKSRC);

DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_BAUDRATE,
                DEBUG_CONSOLE_DEVICE_TYPE_UART, uartClkSrcFreq);
```

## Function groups

### 28.2.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype "`%[flags][width][.precision][length]specifier`", which is explained below

flags	Description
-	Left-justified within the given field width. Right-justified is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is written, a blank space is inserted before the value.
#	Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.



<b>.precision</b>	<b>Description</b>
.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.

<b>length</b>	<b>Description</b>
Do not support	

<b>specifier</b>	<b>Description</b>
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 */
```

## 28.3 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\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\r", "1 day", 86400, 86.4);
```

### Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

### Print out failure messages using MCUXpresso SDK \_\_assert\_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \" %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file ,
        line, func);
    for (;;)
    {}
}
```

### Note:

To use 'printf' and 'scanf' for GNUC Base, add file 'fsl\_sbrk.c' in path: ..\{package}\devices\{subset}\utilities\fsl-\_sbrk.c to your project.

## Modules

- [Semihosting](#)

## Data Structures

- struct [io\\_state\\_t](#)  
*State structure storing io. [More...](#)*

## Macros

- #define **SDK\_DEBUGCONSOLE** 1U  
*Definition to select sdk or toolchain printf, scanf.*

## Typedefs

- typedef void(\* **notify**)(size\_t \*size, bool rx, bool tx)  
*define a notify callback for IO*
- typedef void(\* **printfCb**)(char \*buf, int32\_t \*indicator, char val, int len)  
*A function pointer which is used when format printf log.*

## Functions

- void **IO\_Init**(io\_state\_t \*io, uint32\_t baudRate, uint32\_t clkSrcFreq, uint8\_t \*ringBuffer)  
*io init function.*
- status\_t **IO\_Deinit**(void)  
*Deinit IO.*
- status\_t **IO\_Transfer**(uint8\_t \*ch, size\_t size, bool tx)  
*io transfer function.*
- status\_t **IO\_WaitIdle**(void)  
*io wait idle.*
- status\_t **LOG\_Init**(uint32\_t baseAddr, uint8\_t device, uint32\_t baudRate, uint32\_t clkSrcFreq)  
*Initializes.*
- void **LOG\_Deinit**(void)  
*De-Initializes.*
- int **LOG\_Push**(uint8\_t \*buf, size\_t size)  
*log push interface*
- int **LOG\_ReadLine**(uint8\_t \*buf, size\_t size)  
*log read one line function*
- int **LOG\_ReadCharacter**(uint8\_t \*ch)  
*log read one character function*
- status\_t **LOG\_WaitIdle**(void)  
*wait log and io idle*
- int **LOG\_Pop**(uint8\_t \*buf, size\_t size)  
*log pop function*
- 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**(uint32\_t baseAddr, uint32\_t baudRate, uint8\_t device, uint32\_t clkSrcFreq)  
*Initializes the the peripheral used for debug messages.*
- status\_t **DbgConsole\_Deinit**(void)  
*De-initializes the peripheral used for debug messages.*
- int **DbgConsole\_Printf**(const char \*fmt\_s,...)  
*Writes formatted output to the standard output stream.*
- int **DbgConsole\_Putchar**(int ch)

## Function Documentation

- Writes a character to stdout.*
- int [DbgConsole\\_Scanf](#) (char \*fmt\_ptr,...)  
*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 log.*

## 28.4 Data Structure Documentation

### 28.4.1 struct io\_state\_t

#### Data Fields

- void \* [ioBase](#)  
*Base of the IP register.*
- uint8\_t [ioType](#)  
*device type*

#### 28.4.1.0.0.27 Field Documentation

##### 28.4.1.0.0.27.1 void\* io\_state\_t::ioBase

## 28.5 Macro Definition Documentation

### 28.5.1 #define SDK\_DEBUGCONSOLE 1U

## 28.6 Typedef Documentation

### 28.6.1 typedef void(\* notify)(size\_t \*size, bool rx, bool tx)

Parameters

<i>size,transfer</i>	data size.
<i>rx,indicate</i>	a rx transfer is success.
<i>tx,indicate</i>	a tx transfer is success.

## 28.7 Function Documentation

### 28.7.1 status\_t DbgConsole\_Init ( uint32\_t baseAddr, uint32\_t baudRate, uint8\_t device, uint32\_t clkSrcFreq )

Call this function to enable debug log messages to be output via the specified peripheral, frequency of peripheral source clock, and base address at the specified baud rate. After this function has returned, stdout and stdin are connected to the selected peripheral.

## Parameters

<i>baseAddr</i>	Indicates the address of the peripheral used to send debug messages.
<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"> <li>• <code>DEBUG_CONSOLE_DEVICE_TYPE_UART</code>,</li> <li>• <code>DEBUG_CONSOLE_DEVICE_TYPE_LPUART</code>,</li> <li>• <code>DEBUG_CONSOLE_DEVICE_TYPE_LPSCI</code>,</li> <li>• <code>DEBUG_CONSOLE_DEVICE_TYPE_USBCDC</code>.</li> </ul>
<i>clkSrcFreq</i>	Frequency of peripheral source clock.

## Returns

Indicates whether initialization was successful or not.

## Return values

<i>kStatus_Success</i>	Execution successfully
<i>kStatus_Fail</i>	Execution failure
<i>kStatus_InvalidArgument</i>	Invalid argument existed

### 28.7.2 `status_t DbgConsole_Deinit ( void )`

Call this function to disable debug log messages to be output via the specified peripheral base address and at the specified baud rate.

## Returns

Indicates whether de-initialization was successful or not.

### 28.7.3 `int DbgConsole_Printf ( const char * fmt_s, ... )`

Call this function to write a formatted output to the standard output stream.

## Function Documentation

### Parameters

<i>fmt_s</i>	Format control string.
--------------	------------------------

### Returns

Returns the number of characters printed or a negative value if an error occurs.

## 28.7.4 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.

## 28.7.5 int DbgConsole\_Scanf ( char \* *fmt\_ptr*, ... )

Call this function to read formatted data from the standard input stream.

### Parameters

<i>fmt_ptr</i>	Format control string.
----------------	------------------------

### Returns

Returns the number of fields successfully converted and assigned.

## 28.7.6 int DbgConsole\_Getchar ( void )

Call this function to read a character from standard input.

### Returns

Returns the character read.



### 28.7.7 `status_t DbgConsole_Flush ( void )`

Call this function to wait the buffer empty and io idle before. 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.

### 28.7.8 `void IO_Init ( io_state_t * io, uint32_t baudRate, uint32_t clkSrcFreq, uint8_t * ringBuffer )`

Call this function to init IO.

Parameters

<i>io</i>	configuration pointer
<i>baudRate</i>	baud rate
<i>clkSrcFreq</i>	clock freq
<i>ringbuffer</i>	used to receive character

### 28.7.9 `status_t IO_Deinit ( void )`

Call this function to Deinit IO.

Returns

deinit status

### 28.7.10 `status_t IO_Transfer ( uint8_t * ch, size_t size, bool tx )`

Call this function to transfer log. Print log:

```
* IO_Transfer(ch, size, true);
*
```

Scanf log:

```
* IO_Transfer(ch, size, false);
*
```

## Function Documentation

### Parameters

<i>ch</i>	transfer buffer pointer
<i>size</i>	transfer size
<i>tx</i>	indicate the transfer is TX or RX

### 28.7.11 `status_t IO_WaitIdle ( void )`

Call this function to wait the io idle

### Returns

Indicates whether wait idle was successful or not.

### 28.7.12 `status_t LOG_Init ( uint32_t baseAddr, uint8_t device, uint32_t baudRate, uint32_t clkSrcFreq )`

Call this function to init the buffer

### Parameters

<i>base-Addr,device</i>	base address
<i>device,device</i>	type
<i>baud-Rate,device</i>	communicate baudrate
<i>clkSrc-Freq,device</i>	source clock freq

### Returns

Indicates whether initialization was successful or not.

### Return values

---

<i>kStatus_Success</i>	Execution successfully
<i>kStatus_Fail</i>	Execution failure

### 28.7.13 void LOG\_Deinit ( void )

Call this function to deinit the buffer

Returns

Indicates whether Deinit was successful or not.

### 28.7.14 int LOG\_Push ( uint8\_t \* *buf*, size\_t *size* )

Call this function to print log

Parameters

<i>fmt,buffer</i>	pointer
<i>size,available</i>	size

Returns

indicate the push size

Return values

	indicate buffer is full or transfer fail.
<i>size</i>	return the push log size.

### 28.7.15 int LOG\_ReadLine ( uint8\_t \* *buf*, size\_t *size* )

Call this function to print log

Parameters

## Function Documentation

<i>fmt,buffer</i>	pointer
<i>size,available</i>	size the number of the recieved character

### 28.7.16 int LOG\_ReadCharacter ( uint8\_t \* *ch* )

Call this function to GETCHAR

Parameters

<i>ch</i>	receive address the number of the recieved character
-----------	--

### 28.7.17 status\_t LOG\_WaitIdle ( void )

Call this function to wait log buffer empty and io idle before enter low power mode.

Returns

Indicates whether wait idle was successful or not.

### 28.7.18 int LOG\_Pop ( uint8\_t \* *buf*, size\_t *size* )

Call this function to pop log from buffer.

Parameters

<i>buf</i>	buffer address to pop
<i>size</i>	log size to pop

Returns

pop log size.

### 28.7.19 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_ptr</i>	Format string for printf.
in	<i>args_ptr</i>	Arguments to printf.
in	<i>buf</i>	pointer to the buffer
	<i>cb</i>	print callbck function pointer

## Returns

Number of characters to be print

### 28.7.20 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 "".
---------------	-----------------------------------

### 28.8 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.

#### 28.8.1 Guide Semihosting for IAR

**NOTE:** After the setting both "printf" and "scanf" are available for debugging.

##### Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

##### Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

##### Step 3: Starting semihosting

1. Choose "Semihosting\_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Start the project by choosing Project>Download and Debug.
4. Choose View>Terminal I/O to display the output from the I/O operations.

#### 28.8.2 Guide Semihosting for Keil µVision

**NOTE:** Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

##### Step 1: Prepare code

Remove function `fputc` and `fgetc` is used to support KEIL in "fsl\_debug\_console.c" and add the following code to project.

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;    /* used for Debug Input */
```

```

struct __FILE
{
    int handle;
};
FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
    return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{
    /* blocking */
    while (ITM_CheckChar() != 1)
        ;
    return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}

void _ttywrch(int ch)
{
    ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
    goto label; /* endless loop */
}

```

## Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click.
2. Select "Target" tab and not select "Use MicroLIB".
3. Select "Debug" tab, select "J-Link/J-Trace Cortex" and click "Setting button".
4. Select "Debug" tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK.

## Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

## Step 4: Building the project

1. Choose "Debug" on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

### 28.8.3 Guide Semihosting for KDS

**NOTE:** After the setting use "printf" for debugging.

#### Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select "Libraries" on "Cross ARM C Linker" and delete "nosys".
3. Select "Miscellaneous" on "Cross ARM C Linker", add "-specs=rdimon.specs" to "Other link flages" and tick "Use newlib-nano", and click OK.

#### Step 2: Building the project

1. In menu bar, choose Project>Build Project.

#### Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick "Enable semihosting and Telnet". Press "Apply" and "Debug".
2. After clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

### 28.8.4 Guide Semihosting for MCUX

#### 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.



## 28.8.5 Guide Semihosting for ARMGCC

### Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
  - "Host Name (or IP address)" : localhost
  - "Port" :2333
  - "Connection type" : Telet.
  - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

#### Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__stack_size__=0x2000")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
defsym=__stack_size__=0x2000")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
defsym=__heap_size__=0x2000")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__heap_size__=0x2000")
```

### Step 2: Building the project

1. Change "CMakeLists.txt":
 

**Change** "SET(CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE "\${CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE} -specs=nano.specs")"

**to** "SET(CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE "\${CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE} -specs=rdimon.specs")"

**Replace paragraph**

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mapcs")
```

## Semihosting

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} --gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} -static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} -z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} muldefs")
```

### To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBU-
G} --specs=rdimon.specs ")
```

### Remove

```
target_link_libraries(semihosting_ARMGCC.elf debug nosys)
```

2. Run "build\_debug.bat" to build project

## Step 3: Starting semihosting

- (a) Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\trkr64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

- (b) After the setting, press "enter". The PuTTY window now shows the printf() output.

**How to Reach Us:****Home Page:**[nxp.com](http://nxp.com)**Web Support:**[nxp.com/support](http://nxp.com/support)

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

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

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

© 2018 NXP B.V.

Document Number: MCUXSDKLPC51U68APIRM

Rev. 0

Feb 2018

