

Kinetis SDK API Reference Manual

Freescale Semiconductor, Inc.

KSDKAPIRM
1.0.0
Jul 2014

Contents

Chapter Introduction

Chapter Architectural Overview

Chapter Analog-to-Digital Converter (ADC)

3.1	ADC HAL Driver	14
3.1.1	Enumeration Type Documentation	16
3.1.2	Function Documentation	18
3.2	ADC Peripheral Driver	26
3.2.1	Data Structure Documentation	32
3.2.2	Function Documentation	34

Chapter Controller Area Network (FlexCAN)

4.1	FlexCAN HAL driver	42
4.1.1	Data Structure Documentation	47
4.1.2	Enumeration Type Documentation	49
4.1.3	Function Documentation	52
4.2	FlexCAN Driver	69
4.2.1	Data Structure Documentation	72
4.2.2	Enumeration Type Documentation	73
4.2.3	Function Documentation	73

Chapter Comparator (CMP)

5.1	CMP HAL Driver	80
5.1.1	Enumeration Type Documentation	82
5.1.2	Function Documentation	83
5.2	CMP Peripheral Driver	92
5.2.1	Data Structure Documentation	96
5.2.2	Enumeration Type Documentation	98
5.2.3	Function Documentation	99

Contents

Section Number	Title	Page Number
Chapter	Digital-to-Analog Converter (DAC)	
6.1	DAC HAL Driver	106
6.1.1	Enumeration Type Documentation	107
6.1.2	Function Documentation	109
6.2	DAC Peripheral Driver	117
6.2.1	Data Structure Documentation	122
6.2.2	Enumeration Type Documentation	124
6.2.3	Function Documentation	124
Chapter	Direct Memory Access (DMA)	
7.1	DMA HAL driver	132
7.1.1	Data Structure Documentation	134
7.1.2	Enumeration Type Documentation	134
7.1.3	Function Documentation	135
7.2	DMAMUX HAL driver	144
7.3	Enumeration Type Documentation	144
7.3.1	dmamux_dma_request_source	144
7.4	Function Documentation	144
7.4.1	DMAMUX_HAL_Init	144
7.4.2	DMAMUX_HAL_SetChannelCmd	144
7.4.3	DMAMUX_HAL_SetPeriodTriggerCmd	145
7.4.4	DMAMUX_HAL_SetTriggerSource	145
7.5	DMA Driver	146
7.5.1	Data Structure Documentation	149
7.5.2	Typedef Documentation	149
7.5.3	Enumeration Type Documentation	149
7.5.4	Function Documentation	150
7.6	DMA request	155
Chapter	Serial Peripheral Interface (DSPI)	
8.1	DSPI HAL driver	158
8.1.1	Data Structure Documentation	162
8.1.2	Enumeration Type Documentation	164
8.1.3	Function Documentation	167
8.2	DSPI Master Driver	184
8.2.1	Data Structure Documentation	188

Contents

Section Number	Title	Page Number
8.2.2	Function Documentation	190
8.3	DSPI Slave Driver	197
8.3.1	Data Structure Documentation	199
8.3.2	Function Documentation	201
8.4	DSPI Shared IRQ Driver	204
8.4.1	Function Documentation	204
Chapter	Enhanced Direct Memory Access (eDMA)	
9.1	eDMA HAL driver	206
9.1.1	Data Structure Documentation	211
9.1.2	Enumeration Type Documentation	213
9.1.3	Function Documentation	214
9.2	eDMA Peripheral Driver	242
9.2.1	Data Structure Documentation	247
9.2.2	Macro Definition Documentation	248
9.2.3	Typedef Documentation	248
9.2.4	Enumeration Type Documentation	249
9.2.5	Function Documentation	249
9.3	eDMA request	259
Chapter	Ethernet MAC (ENET)	
10.1	ENET HAL driver	262
10.1.1	Data Structure Documentation	273
10.1.2	Macro Definition Documentation	279
10.1.3	Enumeration Type Documentation	279
10.1.4	Function Documentation	285
10.2	ENET Peripheral Driver	338
10.2.1	Data Structure Documentation	347
10.2.2	Macro Definition Documentation	351
10.2.3	Enumeration Type Documentation	351
10.2.4	Function Documentation	351
10.3	ENET RTCS Adaptor	358
10.3.1	Data Structure Documentation	360
10.3.2	Function Documentation	362
10.4	ENET Physical Layer Driver	369

Contents

Section Number	Title	Page Number
Chapter	FlexTimer (FTM)	
11.1	FlexTimer HAL driver	372
11.1.1	Data Structure Documentation	377
11.1.2	Macro Definition Documentation	378
11.1.3	Enumeration Type Documentation	378
11.1.4	Function Documentation	379
11.2	FlexTimer Peripheral Driver	411
11.2.1	Data Structure Documentation	412
11.2.2	Function Documentation	412
Chapter	General Purpose Input/Output (GPIO)	
12.1	GPIO HAL driver	418
12.1.1	Enumeration Type Documentation	419
12.1.2	Function Documentation	419
12.2	GPIO Peripheral Driver	425
12.2.1	Data Structure Documentation	429
12.2.2	Macro Definition Documentation	431
12.2.3	Function Documentation	431
12.3	GPIO ISR Definitions	437
Chapter	Inter-Integrated Circuit (I2C)	
13.1	I2C HAL driver	440
13.1.1	Enumeration Type Documentation	443
13.1.2	Function Documentation	443
13.2	I2C Slave peripheral driver	455
13.2.1	Data Structure Documentation	455
13.2.2	Function Documentation	455
13.3	I2C Master peripheral	457
13.3.1	Data Structure Documentation	458
13.3.2	Function Documentation	459
13.4	I2C Classes	463
Chapter	Low Power Timer (LPTMR)	
14.1	LPTMR HAL driver	466
14.1.1	Enumeration Type Documentation	468
14.1.2	Function Documentation	470

Contents

Section Number	Title	Page Number
14.2	LPTMR Peripheral Driver	478
14.2.1	Data Structure Documentation	479
14.2.2	Function Documentation	481
Chapter Low Power Universal Asynchronous Receiver/Transmitter (LPUART)		
15.1	LPUART HAL driver	486
15.1.1	Data Structure Documentation	492
15.1.2	Enumeration Type Documentation	492
15.1.3	Function Documentation	496
15.2	LPUART peripheral Driver	513
15.2.1	Data Structure Documentation	515
15.2.2	Typedef Documentation	517
15.2.3	Function Documentation	517
15.3	LPUART Types Definitions	523
Chapter Memory Protection Unit (MPU)		
16.1	MPU HAL driver	526
16.1.1	Enumeration Type Documentation	533
16.1.2	Function Documentation	537
16.2	MPU Peripheral Driver	570
16.2.1	Data Structure Documentation	572
16.2.2	Function Documentation	574
Chapter Programmable Delay Block (PDB)		
17.1	PDB HAL Driver	578
17.1.1	Enumeration Type Documentation	581
17.1.2	Function Documentation	582
17.2	PDB Peripheral Driver	594
17.2.1	Data Structure Documentation	602
17.2.2	Enumeration Type Documentation	605
17.2.3	Function Documentation	605
Chapter Periodic Interrupt Timer (PIT)		
18.1	PIT HAL driver	614
18.1.1	Function Documentation	615
18.2	PIT Peripheral Driver	620

Contents

Section Number	Title	Page Number
18.2.1	Data Structure Documentation	622
18.2.2	Function Documentation	622
18.3	PIT ISR Definitions	627
Chapter Random Number Generator Accelerator (RNGA)		
19.1	RNGA HAL driver	630
19.1.1	Function Documentation	631
19.2	RNGA Peripheral Driver	637
19.2.1	Data Structure Documentation	638
Chapter Real Time Clock (RTC)		
20.1	RTC HAL driver	640
20.1.1	Data Structure Documentation	644
20.1.2	Function Documentation	644
20.2	RTC Peripheral Driver	670
20.2.1	Data Structure Documentation	674
20.2.2	Function Documentation	674
Chapter Synchronous Audio Interface (SAI)		
21.1	SAI HAL driver	682
21.1.1	Enumeration Type Documentation	687
21.1.2	Function Documentation	690
21.2	SAI Peripheral driver	720
21.2.1	Data Structure Documentation	723
21.2.2	Function Documentation	724
Chapter Secured Digital Host Controller (SDHC)		
22.1	SDHC HAL	740
22.1.1	Function Documentation	743
22.2	SDHC Peripheral Driver	774
22.2.1	Function Documentation	775
22.3	SDHC Data Types	778
22.3.1	Data Structure Documentation	780
22.3.2	Enumeration Type Documentation	783
22.4	SDHC Standard Definition	786

Contents

Section Number	Title	Page Number
22.5	SDHC Card Related Standard Definition	794
22.5.1	Enumeration Type Documentation	798
22.6	SDHC Card Definition	801
22.6.1	Data Structure Documentation	801
22.6.2	Enumeration Type Documentation	802
22.7	SDHC Card Driver	803
22.7.1	Function Documentation	803
Chapter	Soundcard (SND)	
23.0.2	Soundcard Driver	809
23.1	Data Structure Documentation	810
23.1.1	struct snd_state_t	810
23.1.2	struct audio_ctrl_operation_t	811
23.1.3	struct audio_codec_operation_t	813
23.1.4	struct audio_controller_t	814
23.1.5	struct audio_codec_t	815
23.1.6	struct audio_buffer_t	815
23.1.7	struct sound_card_t	816
23.2	Macro Definition Documentation	817
23.2.1	USEDMA	817
23.3	Enumeration Type Documentation	817
23.3.1	snd_status_t	817
23.4	Function Documentation	817
23.4.1	SND_TxInit	817
23.4.2	SND_RxInit	817
23.4.3	SND_TxDeinit	818
23.4.4	SND_RxDeinit	818
23.4.5	SND_TxConfigDataFormat	819
23.4.6	SND_RxConfigDataFormat	819
23.4.7	SND_TxUpdateStatus	819
23.4.8	SND_RxUpdateStatus	820
23.4.9	SND_GetStatus	820
23.4.10	SND_TxStart	821
23.4.11	SND_RxStart	821
23.4.12	SND_TxStop	821
23.4.13	SND_RxStop	821
23.4.14	SND_WaitEvent	823
23.4.15	SND_SetMuteCmd	823
23.4.16	SND_SetVolume	823

Contents

Section Number	Title	Page Number
23.4.17	SND_GetVolume	823
Chapter	Serial Peripheral Interface (SPI)	
24.0.18	SPI Master Peripheral Driver	825
24.1	SPI HAL driver	828
24.1.1	Enumeration Type Documentation	830
24.1.2	Function Documentation	831
24.2	SPI Master Peripheral Driver	841
24.2.1	Data Structure Documentation	842
24.2.2	Enumeration Type Documentation	843
24.2.3	Function Documentation	843
24.3	SPI Slave Peripheral Driver	848
24.3.1	Data Structure Documentation	850
24.3.2	Function Documentation	852
24.4	Shared SPI Types	855
24.5	SPI Classes	856
Chapter	Universal Asynchronous Receiver/Transmitter (UART)	
25.1	UART HAL driver	858
25.1.1	Enumeration Type Documentation	862
25.1.2	Function Documentation	865
25.2	UART Peripheral Driver	882
25.2.1	Data Structure Documentation	885
25.2.2	Typedef Documentation	887
25.2.3	Function Documentation	887
Chapter	Watchdog Timer (WDOG)	
26.1	WDOG HAL driver	894
26.1.1	Data Structure Documentation	896
26.1.2	Enumeration Type Documentation	896
26.1.3	Function Documentation	896
26.2	WDOG Peripheral Driver	908
26.2.1	Data Structure Documentation	910
26.2.2	Function Documentation	910

Contents

Section Number	Title	Page Number
Chapter	Low-Leakage Wakeup Unit (LLWU)	
27.0.3	LLWU HAL driver	914
27.1	Data Structure Documentation	914
27.1.1	struct llwu_external_pin_filter_mode_t	914
27.1.2	struct llwu_reset_enable_mode_t	914
27.2	Function Documentation	914
27.2.1	LLWU_HAL_SetExternalInputModuleMode	914
27.2.2	LLWU_HAL_GetExternalInputModuleMode	915
27.2.3	LLWU_HAL_SetInternalModuleCmd	916
27.2.4	LLWU_HAL_GetInternalModuleCmd	916
27.2.5	LLWU_HAL_GetExternalPinWakeUpFlag	916
27.2.6	LLWU_HAL_ClearExternalPinWakeUpFlag	917
27.2.7	LLWU_HAL_GetInternalModuleWakeUpFlag	917
27.2.8	LLWU_HAL_SetPinFilterMode	917
27.2.9	LLWU_HAL_GetPinFilterMode	918
27.2.10	LLWU_HAL_GetFilterDetectFlag	918
27.2.11	LLWU_HAL_ClearFilterDetectFlag	918
Chapter	Multipurpose Clock Generator (MCG)	
28.1	MCG HAL driver	922
28.1.1	Function Documentation	927
Chapter	Oscillator (OSC)	
29.1	OSC HAL driver	962
29.1.1	Enumeration Type Documentation	963
29.1.2	Function Documentation	963
29.2	Shared OSC Types	967
Chapter	Power Management Controller (PMC)	
30.0.1	PMC HAL driver	970
30.1	Enumeration Type Documentation	970
30.1.1	pmc_low_volt_warn_volt_select_t	970
30.1.2	pmc_low_volt_detect_volt_select_t	971
30.1.3	pmc_int_select_t	971
30.2	Function Documentation	971
30.2.1	PMC_HAL_SetLowVoltIntCmd	971
30.2.2	PMC_HAL_SetLowVoltDetectResetCmd	971
30.2.3	PMC_HAL_SetLowVoltDetectAck	972

Contents

Section Number	Title	Page Number
30.2.4	PMC_HAL_GetLowVoltDetectFlag	972
30.2.5	PMC_HAL_SetLowVoltDetectVoltMode	972
30.2.6	PMC_HAL_GetLowVoltDetectVoltMode	973
30.2.7	PMC_HAL_SetLowVoltWarnAck	973
30.2.8	PMC_HAL_GetLowVoltWarnFlag	973
30.2.9	PMC_HAL_SetLowVoltWarnVoltMode	974
30.2.10	PMC_HAL_GetLowVoltWarnVoltMode	974
30.2.11	PMC_HAL_SetBandgapBufferCmd	974
30.2.12	PMC_HAL_GetAckIsolation	975
30.2.13	PMC_HAL_SetClearAckIsolation	975
30.2.14	PMC_HAL_GetRegulatorStatus	975
Chapter	Port Control and Interrupts (PORT)	
31.1	PORT HAL driver	978
31.1.1	Enumeration Type Documentation	980
31.1.2	Function Documentation	981
Chapter	reset control module (RCM)	
32.0.3	RCM HAL driver	987
32.1	Function Documentation	988
32.1.1	RCM_HAL_GetSrcStatusCmd	988
32.1.2	RCM_HAL_SetFilterStopModeCmd	988
32.1.3	RCM_HAL_GetFilterStopModeCmd	988
32.1.4	RCM_HAL_SetFilterRunWaitMode	989
32.1.5	RCM_HAL_GetFilterRunWaitMode	990
32.1.6	RCM_HAL_SetFilterWidth	990
32.1.7	RCM_HAL_GetFilterWidth	990
32.1.8	RCM_HAL_GetEasyPortModeStatusCmd	991
Chapter	System Integration Module (SIM)	
33.1	SIM HAL driver	994
33.1.1	Data Structure Documentation	1004
33.1.2	Function Documentation	1004
Chapter	System Mode Controller (SMC)	
34.1	SMC HAL driver	1054
34.1.1	Data Structure Documentation	1057
34.1.2	Enumeration Type Documentation	1057
34.1.3	Function Documentation	1059

Contents

Section Number	Title	Page Number
Chapter	Clock Manager (Clock)	
35.0.4	Clock Manager	1070
35.1	Enumeration Type Documentation	1071
35.1.1	clock_manager_error_code_t	1071
35.2	Function Documentation	1071
35.2.1	CLOCK_SYS_GetFreq	1071
35.2.2	CLOCK_SYS_SetSource	1072
35.2.3	CLOCK_SYS_GetSource	1072
35.2.4	CLOCK_SYS_SetDivider	1073
35.2.5	CLOCK_SYS_GetDivider	1074
35.2.6	CLOCK_SYS_SetOutDividers	1074
35.2.7	CLOCK_SYS_GetDmaFreq	1074
35.2.8	CLOCK_SYS_GetDmamuxFreq	1075
35.2.9	CLOCK_SYS_GetPortFreq	1075
35.2.10	CLOCK_SYS_GetEwmFreq	1076
35.2.11	CLOCK_SYS_GetFtfFreq	1076
35.2.12	CLOCK_SYS_GetCrcFreq	1077
35.2.13	CLOCK_SYS_GetAdcFreq	1077
35.2.14	CLOCK_SYS_GetCmpFreq	1078
35.2.15	CLOCK_SYS_GetVrefFreq	1078
35.2.16	CLOCK_SYS_GetPdbFreq	1079
35.2.17	CLOCK_SYS_GetFtmFreq	1079
35.2.18	CLOCK_SYS_GetPitFreq	1080
35.2.19	CLOCK_SYS_GetUsbFreq	1080
35.2.20	CLOCK_SYS_GetSpiFreq	1081
35.2.21	CLOCK_SYS_GetI2cFreq	1081
35.2.22	CLOCK_SYS_GetUartFreq	1082
35.2.23	CLOCK_SYS_GetLpuartFreq	1082
35.2.24	CLOCK_SYS_GetSaiFreq	1083
35.2.25	CLOCK_SYS_GetGpioFreq	1083
35.2.26	CLOCK_SYS_EnableDmaClock	1084
35.2.27	CLOCK_SYS_DisableDmaClock	1084
35.2.28	CLOCK_SYS_GetDmaGateCmd	1084
35.2.29	CLOCK_SYS_EnableDmamuxClock	1085
35.2.30	CLOCK_SYS_DisableDmamuxClock	1085
35.2.31	CLOCK_SYS_GetDmamuxGateCmd	1085
35.2.32	CLOCK_SYS_EnablePortClock	1086
35.2.33	CLOCK_SYS_DisablePortClock	1087
35.2.34	CLOCK_SYS_GetPortGateCmd	1087
35.2.35	CLOCK_SYS_EnableEwmClock	1087
35.2.36	CLOCK_SYS_DisableEwmClock	1087
35.2.37	CLOCK_SYS_GetEwmGateCmd	1088
35.2.38	CLOCK_SYS_EnableFtfClock	1088

Contents

Section Number	Title	Page Number
35.2.39	CLOCK_SYS_DisableFtfClock	1088
35.2.40	CLOCK_SYS_GetFtfGateCmd	1088
35.2.41	CLOCK_SYS_EnableCrcClock	1089
35.2.42	CLOCK_SYS_DisableCrcClock	1089
35.2.43	CLOCK_SYS_GetCrcGateCmd	1089
35.2.44	CLOCK_SYS_EnableAdcClock	1090
35.2.45	CLOCK_SYS_DisableAdcClock	1091
35.2.46	CLOCK_SYS_GetAdcGateCmd	1091
35.2.47	CLOCK_SYS_EnableCmpClock	1091
35.2.48	CLOCK_SYS_DisableCmpClock	1091
35.2.49	CLOCK_SYS_GetCmpGateCmd	1092
35.2.50	CLOCK_SYS_EnableDacClock	1092
35.2.51	CLOCK_SYS_DisableDacClock	1092
35.2.52	CLOCK_SYS_GetDacGateCmd	1092
35.2.53	CLOCK_SYS_EnableVrefClock	1093
35.2.54	CLOCK_SYS_DisableVrefClock	1093
35.2.55	CLOCK_SYS_GetVrefGateCmd	1093
35.2.56	CLOCK_SYS_EnableSaiClock	1094
35.2.57	CLOCK_SYS_DisableSaiClock	1095
35.2.58	CLOCK_SYS_GetSaiGateCmd	1095
35.2.59	CLOCK_SYS_EnablePdbClock	1095
35.2.60	CLOCK_SYS_DisablePdbClock	1095
35.2.61	CLOCK_SYS_GetPdbGateCmd	1096
35.2.62	CLOCK_SYS_EnableFtmClock	1096
35.2.63	CLOCK_SYS_DisableFtmClock	1096
35.2.64	CLOCK_SYS_GetFtmGateCmd	1096
35.2.65	CLOCK_SYS_EnablePitClock	1097
35.2.66	CLOCK_SYS_DisablePitClock	1097
35.2.67	CLOCK_SYS_GetPitGateCmd	1097
35.2.68	CLOCK_SYS_EnableLptimerClock	1098
35.2.69	CLOCK_SYS_DisableLptimerClock	1099
35.2.70	CLOCK_SYS_GetLptimerGateCmd	1099
35.2.71	CLOCK_SYS_EnableRtcClock	1099
35.2.72	CLOCK_SYS_DisableRtcClock	1099
35.2.73	CLOCK_SYS_GetRtcGateCmd	1100
35.2.74	CLOCK_SYS_EnableUsbClock	1100
35.2.75	CLOCK_SYS_DisableUsbClock	1100
35.2.76	CLOCK_SYS_GetUsbGateCmd	1100
35.2.77	CLOCK_SYS_EnableSpiClock	1101
35.2.78	CLOCK_SYS_DisableSpiClock	1101
35.2.79	CLOCK_SYS_GetSpiGateCmd	1101
35.2.80	CLOCK_SYS_EnableI2cClock	1102
35.2.81	CLOCK_SYS_DisableI2cClock	1103
35.2.82	CLOCK_SYS_GetI2cGateCmd	1103
35.2.83	CLOCK_SYS_EnableUartClock	1103

Contents

Section Number	Title	Page Number
35.2.84	CLOCK_SYS_DisableUartClock1103
35.2.85	CLOCK_SYS_GetUartGateCmd1104
35.2.86	CLOCK_SYS_EnableLpuartClock1104
35.2.87	CLOCK_SYS_DisableLpuartClock1104
35.2.88	CLOCK_SYS_GetLpuartGateCmd1104
35.2.89	CLOCK_SYS_GetRngaFreq1105
35.2.90	CLOCK_SYS_EnableRngaClock1105
35.2.91	CLOCK_SYS_DisableRngaClock1105
35.2.92	CLOCK_SYS_GetRngaGateCmd1106
35.2.93	CLOCK_SYS_GetFlexbusFreq1106
35.2.94	CLOCK_SYS_EnableFlexbusClock1106
35.2.95	CLOCK_SYS_DisableFlexbusClock1107
35.2.96	CLOCK_SYS_GetFlexbusGateCmd1107
35.2.97	CLOCK_SYS_GetMpuFreq1107
35.2.98	CLOCK_SYS_GetCmtFreq1108
35.2.99	CLOCK_SYS_GetEnetRmiiFreq1108
35.2.100	CLOCK_SYS_GetEnetTimeStampFreq1109
35.2.101	CLOCK_SYS_GetUsbdcdFreq1109
35.2.102	CLOCK_SYS_GetSdhcFreq1110
35.2.103	CLOCK_SYS_EnableMpuClock1110
35.2.104	CLOCK_SYS_DisableMpuClock1110
35.2.105	CLOCK_SYS_GetMpuGateCmd1111
35.2.106	CLOCK_SYS_EnableCmtClock1111
35.2.107	CLOCK_SYS_DisableCmtClock1111
35.2.108	CLOCK_SYS_GetCmtGateCmd1111
35.2.109	CLOCK_SYS_EnableUsbdcdClock1112
35.2.110	CLOCK_SYS_DisableUsbdcdClock1112
35.2.111	CLOCK_SYS_GetUsbdcdGateCmd1112
35.2.112	CLOCK_SYS_EnableFlexcanClock1113
35.2.113	CLOCK_SYS_DisableFlexcanClock1114
35.2.114	CLOCK_SYS_GetFlexcanGateCmd1114
35.2.115	CLOCK_SYS_EnableSdhcClock1114
35.2.116	CLOCK_SYS_DisableSdhcClock1114
35.2.117	CLOCK_SYS_GetSdhcGateCmd1115
35.2.118	CLOCK_SYS_EnableEnetClock1115
35.2.119	CLOCK_SYS_DisableEnetClock1115
35.2.120	CLOCK_SYS_GetEnetGateCmd1115
35.2.121	CLOCK_SYS_GetTpmFreq1116
35.2.122	CLOCK_SYS_EnableTpmClock1116
35.2.123	CLOCK_SYS_DisableTpmClock1116
35.2.124	CLOCK_SYS_GetTpmGateCmd1117
35.2.125	CLOCK_SYS_EnableTsiClock1118
35.2.126	CLOCK_SYS_DisableTsiClock1118
35.2.127	CLOCK_SYS_GetTsiGateCmd1118

Contents

Section Number	Title	Page Number
Chapter	Hwtimer_driver	
36.0.128	HwTimer Driver1119
Chapter	Interrupt Manager (Interrupt)	
37.0.129	Interrupt Manager1121
37.1	Function Documentation1122
37.1.1	INT_SYS_InstallHandler1122
37.1.2	INT_SYS_EnableIRQ1122
37.1.3	INT_SYS_DisableIRQ1122
37.1.4	INT_SYS_EnableIRQGlobal1122
37.1.5	INT_SYS_DisableIRQGlobal1123
Chapter	Power Manager (Power)	
38.0.6	Power Manager1126
38.1	Data Structure Documentation1128
38.1.1	struct power_manager_user_config_t1128
38.1.2	struct power_manager_static_callback_user_config_t1129
38.1.3	struct power_manager_dynamic_callback_user_config_t1129
38.1.4	struct power_manager_state_t1129
38.2	Macro Definition Documentation1130
38.2.1	POWER_SYS_CALLBACK_BEFORE1130
38.2.2	POWER_SYS_CALLBACK_AFTER1130
38.3	Typedef Documentation1130
38.3.1	power_manager_callback_priority_t1130
38.3.2	power_manager_callback_handle_t1130
38.3.3	power_manager_callback_data_t1131
38.3.4	power_manager_callback_t1131
38.4	Enumeration Type Documentation1132
38.4.1	power_manager_modes_t1132
38.4.2	power_manager_error_code_t1132
38.4.3	power_manager_policy_t1132
38.4.4	power_manager_callback_type_t1132
38.5	Function Documentation1133
38.5.1	POWER_SYS_Init1133
38.5.2	POWER_SYS_Deinit1134
38.5.3	POWER_SYS_SetMode1135
38.5.4	POWER_SYS_GetMode1135
38.5.5	POWER_SYS_GetModeConfig1136
38.5.6	POWER_SYS_GetRunningMode1136

Contents

Section Number	Title	Page Number
38.5.7	POWER_SYS_GetErroneousDriver	.1137
38.5.8	POWER_SYS_RegisterCallbackFunction	.1137
38.5.9	POWER_SYS_UnregisterCallbackFunction	.1139
38.5.10	POWER_SYS_GetVeryLowPowerModeStatus	.1139
38.5.11	POWER_SYS_GetLowLeakageWakeupResetStatus	.1140

Chapter Utilities for the Kinetis SDK

39.1	Debug_console	.1142
-------------	-------------------------------	--------------

Chapter OS Abstraction Layer (OSA)

40.0.1	OS Abstraction Layer	.1145
--------	--------------------------------------	-------

40.1	Enumeration Type Documentation	.1151
-------------	--	--------------

40.1.1	osa_status_t	.1151
40.1.2	osa_event_clear_mode_t	.1151
40.1.3	osa_critical_section_mode_t	.1151

40.2	Function Documentation	.1152
-------------	--	--------------

40.2.1	OSA_SemaCreate	.1152
40.2.2	OSA_SemaWait	.1153
40.2.3	OSA_SemaPost	.1154
40.2.4	OSA_SemaDestroy	.1154
40.2.5	OSA_MutexCreate	.1155
40.2.6	OSA_MutexLock	.1156
40.2.7	OSA_MutexUnlock	.1157
40.2.8	OSA_MutexDestroy	.1157
40.2.9	OSA_EventCreate	.1158
40.2.10	OSA_EventWait	.1158
40.2.11	OSA_EventSet	.1159
40.2.12	OSA_EventClear	.1160
40.2.13	OSA_EventDestroy	.1161
40.2.14	OSA_TaskCreate	.1161
40.2.15	OSA_TaskDestroy	.1162
40.2.16	OSA_TaskYield	.1163
40.2.17	OSA_TaskGetHandler	.1163
40.2.18	OSA_TaskGetPriority	.1163
40.2.19	OSA_TaskSetPriority	.1164
40.2.20	OSA_MsgQCreate	.1164
40.2.21	OSA_MsgQPut	.1165
40.2.22	OSA_MsgQGet	.1165
40.2.23	OSA_MsgQDestroy	.1166
40.2.24	OSA_MemAlloc	.1167
40.2.25	OSA_MemAllocZero	.1167
40.2.26	OSA_MemFree	.1167

Contents

Section Number	Title	Page Number
40.2.27	OSA_TimeDelay	1167
40.2.28	OSA_TimeGetMsec	1168
40.2.29	OSA_InstallIntHandler	1168
40.2.30	OSA_EnterCritical	1168
40.2.31	OSA_ExitCritical	1168
40.2.32	OSA_Init	1169
40.2.33	OSA_Start	1169
40.3	Bare Metal Abstraction Layer	1170
40.3.1	Data Structure Documentation	1173
40.3.2	Macro Definition Documentation	1175
40.3.3	Function Documentation	1175
40.4	MQX Abstraction Layer	1177
40.4.1	Macro Definition Documentation	1178
40.4.2	Typedef Documentation	1179
40.5	μC/OS-II Abstraction Layer	1180
40.5.1	Data Structure Documentation	1181
40.5.2	Macro Definition Documentation	1181
40.6	μC/OS-III Abstraction Layer	1184
40.6.1	Data Structure Documentation	1185
40.6.2	Macro Definition Documentation	1185
40.6.3	Typedef Documentation	1187
40.7	FreeRTOS Abstraction Layer	1188
40.7.1	Data Structure Documentation	1189
40.7.2	Macro Definition Documentation	1189
40.7.3	Typedef Documentation	1190
 Chapter Hwtimer_driver		
41.1	Data Structure Documentation	1193
41.1.1	struct hwtimer_t	1193
41.1.2	struct hwtimer_time_t	1194
41.1.3	struct hwtimer_devif_t	1194
41.2	Enumeration Type Documentation	1195
41.2.1	_hwtimer_error_code_t	1195
41.3	Function Documentation	1195
41.3.1	HWTIMER_SYS_Init	1195
41.3.2	HWTIMER_SYS_Deinit	1196
41.3.3	HWTIMER_SYS_SetFreq	1196
41.3.4	HWTIMER_SYS_SetPeriod	1197

Contents

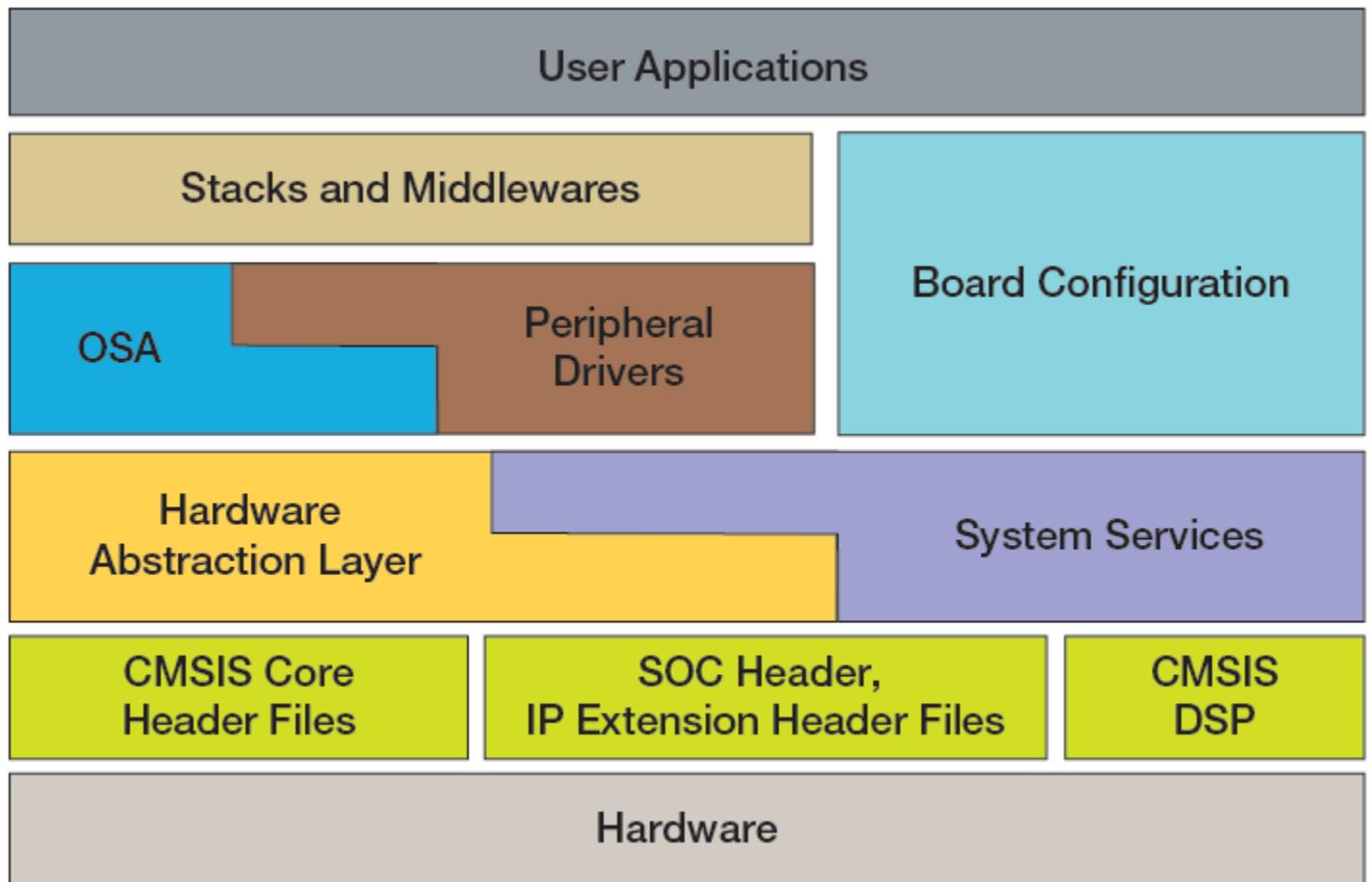
Section Number	Title	Page Number
41.3.5	HWTIMER_SYS_GetFreq1198
41.3.6	HWTIMER_SYS_GetPeriod1198
41.3.7	HWTIMER_SYS_Start1199
41.3.8	HWTIMER_SYS_Stop1199
41.3.9	HWTIMER_SYS_GetModulo1200
41.3.10	HWTIMER_SYS_GetTime1201
41.3.11	HWTIMER_SYS_GetTicks1202
41.3.12	HWTIMER_SYS_RegisterCallback1203
41.3.13	HWTIMER_SYS_BlockCallback1204
41.3.14	HWTIMER_SYS_UnblockCallback1205
41.3.15	HWTIMER_SYS_CancelCallback1205

Chapter 1

Introduction

The Kinetis software development kit (KSDK) is an extensive suite of robust peripheral drivers, stacks and middleware designed to simplify and accelerate application development on any Freescale Kinetis MCU. The addition of Processor Expert technology for software and board support configuration provides unmatched ease of use and flexibility. The Kinetis SDK is complimentary and includes full source code under a permissive open-source license for all hardware abstraction and peripheral driver software.

The Kinetis SDK consists of the following runtime software components written in C:



- ARM CMSIS Core and DSP standard libraries and CMSIS-compliant device header files which provide direct access to the peripheral registers and bits
- An open-source hardware abstraction layer (HAL) that provides a simple, stateless driver with an API encapsulating the low-level functions of the peripheral
- System services for centralized resources including a clock manager, interrupt manager, low-power manager, and a hardware timer

- Open-source, high-level peripheral drivers that build upon the HAL layer and may utilize one or more of the system services; drivers may be used as-is or as a reference for creating custom drivers
- An operating system abstraction (OSA) layer for adapting applications for use with a real time operating system (RTOS) or bare metal (no RTOS) applications. OSAs are provided for:
 - Freescale MQX™ RTOS
 - FreeRTOS
 - Micrium uC/OS-II
 - Micrium uC/OS-III
 - CMSIS-RTOS API compliant RTOS
 - bare-metal (no RTOS)
- Stacks and middleware in source or object formats including:
 - a comprehensive device and host USB stack with comprehensive USB class support
 - CMSIS DSP, a suite of common signal processing functions
 - FatFs, a FAT file system for small embedded systems
 - Encryption software utilizing the mmCAU hardware acceleration unit

The Kinetis SDK comes complete with software examples demonstrating the usage of the HAL, peripheral drivers, middleware, and RTOSes. All examples are provided with projects for the following toolchains:

- IAR Embedded Workbench
- Keil MDK
- Kinetis Design Studio
- GNU toolchain for ARM® Cortex®-M with makefile system

The HAL, peripheral drivers and system services can be used across multiple devices within the Kinetis product family without code modification. The configurable items for each driver, at all levels, are encapsulated into C language data structures. Kinetis devices specific configuration information is provided as part of the SDK and need not be modified by the user. HAL, peripheral driver, and system services configuration is not fixed and can be changed at runtime. Optionally, the entire driver set can be pre-built into a library.

The example applications demonstrate how to configure the drivers by passing configuration data to the APIs. In addition to the software source, Processor Expert is provided as a time-saving option for software configuration. Processor Expert is a complimentary PC-hosted software configuration tool (Eclipse plugin) with complete knowledge of all Kinetis MCUs. It provides a graphical user interface to handle MCU-specific board configuration and driver tuning tasks including:

- Optional generation of low-level device initialization code for post-reset configuration
- Package I/O allocation and pin initialization source code generation
- Creation and management of HAL and peripheral driver C source configuration data structures

The organization of files in the Kinetis SDK release package is focused on ease-of-use. The Kinetis SDK folder hierarchy is organized at the top level with these folders:

platform	Platform folder contains code for SDK HAL and Peripheral drivers. The folder contain following sub-folders								
drivers	<p>Peripheral drivers for every peripheral supported For each peripheral there is a sub-folder under it containing following files/folders:</p> <table border="1"> <tr> <td>Source</td><td>Source folder containing .c files for the driver code</td></tr> <tr> <td>fsl_ipname_driver.h</td><td>Peripheral driver API header file</td></tr> </table>			Source	Source folder containing .c files for the driver code	fsl_ipname_driver.h	Peripheral driver API header file		
Source	Source folder containing .c files for the driver code								
fsl_ipname_driver.h	Peripheral driver API header file								
hal	<p>HAL drivers for every peripheral supported For each peripheral there is a sub-folder under it containing following files:</p> <table border="1"> <tr> <td>fsl_ipname_hal.c</td><td>HAL driver source code file</td></tr> <tr> <td>fsl_ipname_hal.h</td><td>HAL driver API header file</td></tr> <tr> <td>fsl_ipname_features.h</td><td>Features header file for the IP defining features supported for specific chipset</td></tr> </table>			fsl_ipname_hal.c	HAL driver source code file	fsl_ipname_hal.h	HAL driver API header file	fsl_ipname_features.h	Features header file for the IP defining features supported for specific chipset
fsl_ipname_hal.c	HAL driver source code file								
fsl_ipname_hal.h	HAL driver API header file								
fsl_ipname_features.h	Features header file for the IP defining features supported for specific chipset								
include	Platform specific headers and board specific CMSIS header files								
startup	Chip specific CMSIS compliant startup code								
utilities	Tools								
linker	Board specific linker files								
apps	Contain demo applications code								
doc	User Guide and Quick start guides for each chip supported								
boards	Board specific code								
lib	Prebuilt libraries for each tool chain and boards supported								
mk	Common make files used for compiling with GCC								
rtos	RTOS abstraction layer code for supported RTOSes								
usb	Freescale's USB stack code								

The rest of the document describes the API references in detail for HAL and Peripheral drivers.

Chapter 2

Architectural Overview

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

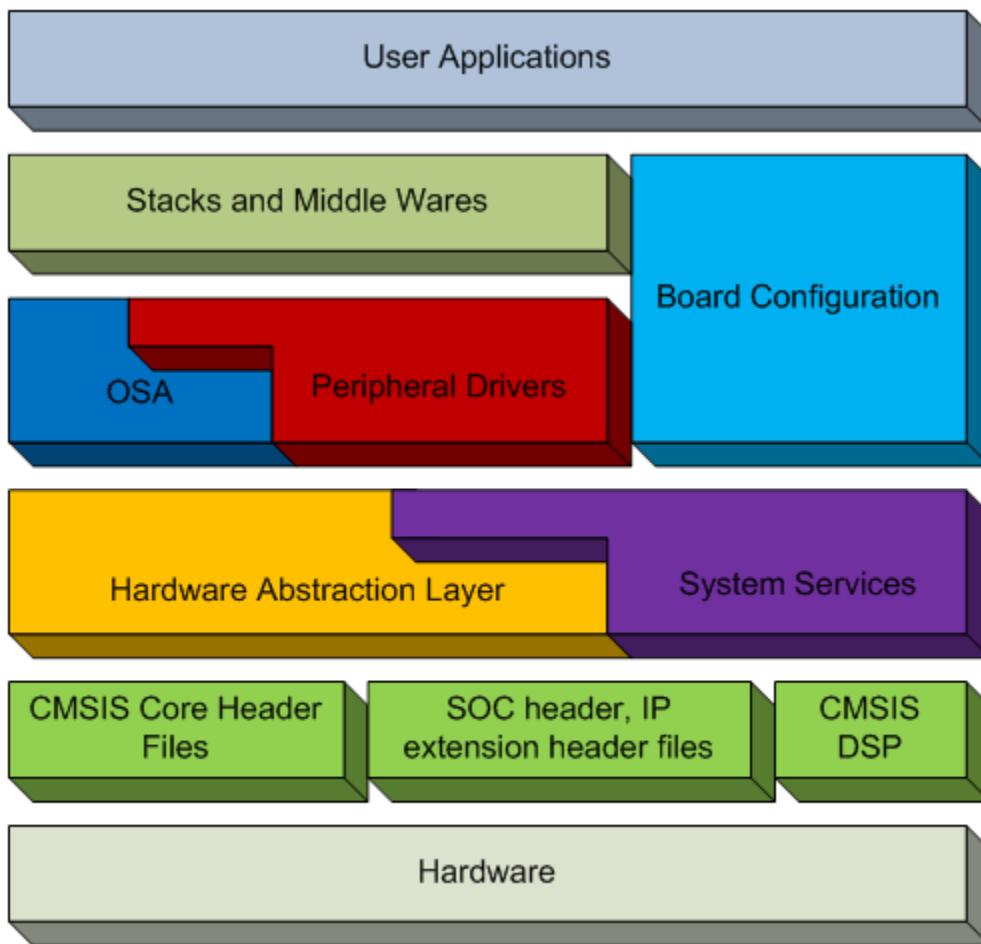
Overview

The KSDK architecture consists of eight key components listed below.

1. The ARM Cortex Microcontroller Software Interface Standard (CMSIS) CORE compliance device specific header files
2. System Services
3. Hardware Abstraction Layer
4. Peripheral Driver Layer
5. Real-time Operating System (RTOS) Abstraction Layer
6. Board-specific configuration
7. Stack and Middleware that integrate with KSDK
8. Applications based on the KSDK architecture

This image shows how each component stacks up.

The Kinetis SDK consists of these runtime software components written in C:



Kinetis MCU header files

The KSDK contains CMSIS-compliant device-specific header files which provide direct access to the Kinetis MCU peripheral registers. Each supported Kinetis MCU device in KSDK has an overall System-on-Chip (SoC) memory-mapped header file. This header file contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers. The overall SoC header file provides an access to the peripheral registers through pointers and predefined masks.

In addition to the overall SoC memory-mapped header file, the KSDK includes extension header files for each peripheral instantiated on the Kinetis MCU. These peripheral extension header files define C union data structures that represent each register bit field within a peripheral for convenient register accesses. The extension header files also take advantage of the bit band feature when reading from or writing to 1-bit wide fields within a register. It also provides register address offsets for each instance of the peripheral along with address offset for each register within an instance. When accessing the register macros in the extension header file, the user must pass in the register base address for the desired peripheral. The KSDK HAL Driver (discussed later) API functions take in the base address and then passes this into the extension header file macros for peripheral register accesses.

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

files can all be found in the KSDK platform/CMSIS directory.

System Services

The KSDK System Services contains a set of software entities that can be used by the Peripheral Drivers. They may be used with HAL Drivers to build the Peripheral Drivers or they can be used by an application directly. The following sections describe each of the System Services software entities. These System Services are in the KSDK platform/system directory.

Interrupt Manager

The Interrupt Manager provides functions to enable and disable individual interrupts within the Nested Vector Interrupt Controller (NVIC). It also provides functions to enable and disable the ARM core global interrupt (via the CPSIE and CPSID instructions) for bare-metal critical section implementation. In addition to providing functions for interrupt enabling and disabling, the Interrupt Manager provides Interrupt Service Routine (ISR) registration that allows the application software to register or replace the interrupt handler for a specified IRQ vector. The KSDK drivers do not set interrupt priorities. The interrupt priority scheme is entirely determined by the specific application logic and its setting is handled by the user application. The user application manages the interrupt priorities by using the NVIC functions provided in the ARM Cortex-M core CMSIS header file.

Clock Manager

The Clock Manager provides centralized clock-related functions for the entire system. It can dynamically set the system clock and perform clock gating/un-gating for specific peripherals. The Clock Manager also maintains knowledge of the clock sources required for each peripheral and provides functions to obtain the clock frequency for each supported clock used by the peripheral.

Unified Hardware (HW) Timer

The Unified HW Timer provides a common timer interface that can be linked with any supported Kinetis MCU hardware timer peripheral or with the ARM core system timer (SysTick) to perform basic timer operations. It can be used as a shared timer capable of microsecond resolution for bare-metal use-cases, such as to time guard busy loops. It can also be used when interfacing with an RTOS to provide operating system (OS) time ticks. Future implementations of the Unified HW Timer will take advantage of low power timer peripherals that will allow OS ticks to continue when the system is in various lower power modes.

Hardware Abstraction Layer (HAL)

The KSDK HAL consists of low-level drivers for the Kinetis MCU product family on-chip peripherals. The main goal is to abstract the hardware peripheral register accesses into a set of stateless basic functional operations. The HAL itself can be used with system services to build application-specific logic or as building blocks for use-case driven high-level Peripheral Drivers. It primarily focuses on the functional control, configuration, and realization of basic peripheral operations. The HAL hides register access details and various MCU peripheral instantiation differences so that, either an application or high-level Peripheral Drivers, can be abstracted from the low-level HW details. Therefore, hardware peripheral must be accessed through HAL.

The HAL can also support some high-level functions with certain logic, provided these high-level functions do not depend on functions from other peripherals, nor impose any action to be taken in interrupt

service routines. For example, the UART HAL provides a blocking byte-send function that relies only on the features available in the UART peripheral itself. These high-level functions enhance the usability of the HAL but remain stateless and independent of other peripherals. Essentially, the HAL functional boundary is limited by the peripheral itself. There is one HAL driver for each peripheral and the HAL only accesses the features available within the peripheral. In addition, the HAL does not define interrupt service routine entries or support interrupt handling. These tasks must be handled by a high-level Peripheral Driver together with the Interrupt Manager.

The HAL drivers can be found in the KSDK platform/hal directory.

Feature Header Files

The HAL is designed to be reusable regardless of the peripheral configuration differences from one Kinetis MCU device to another. A Peripheral Feature Header File is provided for each peripheral type to define the feature or configuration differences for each Kinetis sub-family device. The feature header files are integrated with the HAL so that it is tailored to the exact feature or configuration supported for a particular device.

Design Guidelines

This section summarizes the design guidelines to develop the HAL drivers. It is meant for information purposes and provides more details on the make-up of the HAL drivers. As previously stated, the main goal of the HAL is to abstract the hardware details and provide a set of easy-to-use low-level drivers. The HAL itself can be used directly by the user application; in this case, the high-level Peripheral Drivers that are built on top of the HAL serve as a reference implementation and to demonstrate the HAL usage. The HAL is mainly focused on individual functional primitives and makes no assumption of use-cases. It also implements some high-level functions with certain logic without dependencies from other peripherals and does not impose any interrupt handling. The HAL APIs follow a naming convention that is consistent from one peripheral to the next. This is a summary of the design guidelines used when developing the HAL drivers:

- There is a dedicated HAL driver for each individual peripheral. Peripherals with different versions or configurations are treated as the same peripheral with differences handled through a feature header file. HAL should hide both the peripheral and MCU details and differences from the high-level application and drivers.
- Each HAL driver has an initialization function to put the peripheral into a known default state (i.e. the peripheral reset state). There is no corresponding de-initialization function.
- Each HAL driver has an enable and disable function to enable or disable the peripheral module.
- The HAL driver does not have an internal operation context and should not dynamically allocate memory.
- The HAL driver does not take in any configuration data from a high-level driver or high-level application because the HAL does not assume any use-cases.
- The HAL provides both blocking and non-blocking functions for peripherals that support data transaction-related operations.
- The HAL may implement high-level functions based on abstracted functional primitives, provided this implementation does not depend on functions outside of the peripheral and does not involve interrupt handling. The HAL must remain stateless.
- The HAL driver does not depend on any other software entities or invoke any functions from other

peripherals.

Peripheral Drivers

The KSDK Peripheral Drivers are use-case driven high-level drivers that implement high-level logic transactions based on one or more HAL drivers, other Peripheral Drivers, and/or System Services. Consider, for example, the differences in the UART HAL and the UART Peripheral Driver. The UART HAL mainly focuses on byte-level basic functional primitives, while the UART Peripheral Driver operates on an interrupt-driven level using data buffers to transfer a stream of bytes and may be able to interface with DMA Peripheral Driver for DMA-enabled transfers between data buffers. In general, if a driver, that is mainly based on one peripheral, interfaces with functions beyond its own HAL and/or requires interrupt servicing, the driver is considered a high-level Peripheral Driver.

The KSDK Peripheral Drivers support all instances of each peripheral instantiated on the Kinetis MCU by using a simple integer parameter for the peripheral instance number. Each Peripheral Driver includes a “common” file, denoted as `fsl_<peripheral>_common.c` (where `<peripheral>` is the name of the peripheral for which the driver is written). This file contains the translation of the peripheral instance number to the peripheral register base address, which is then passed into the HAL. Hence, the user of the Peripheral Driver does not need to know the peripheral memory-mapped base address.

The Peripheral Drivers operate on a 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 for the driver internal operation through the driver initialization function.

The Peripheral Drivers are designed to handle the entire functionality for a targeted use-case. An application should be able to use only the Peripheral Driver to accomplish its purpose. The mixing of the Peripheral Driver and HAL by an application for the same peripheral can be done, but is discouraged for architectural cleanliness and to avoid cases where bypassing the Peripheral Driver results in logic errors within the Peripheral Driver.

The Peripheral Drivers can be found in the KSDK platform/drivers directory.

Interrupt Handling

Interrupt-driven Peripheral Drivers are designed to service all interrupts for their desired functionality. Each Peripheral Driver implements a set of easy-to-use APIs for a particular use-case. Each Peripheral Driver also exposes interrupt functions that implement all actions taken when serving a particular interrupt. These interrupt action functions take in the peripheral instance number as an input parameter. All Interrupt Service Routine (ISR) entries for the Peripheral Driver are implemented in a separate C file named the `fsl_<peripheral>_irq.c`, which is located in the top level of the Peripheral Driver folder. The ISR entries invoke a corresponding interrupt action functions that are implemented in the driver and are exposed as part of the driver public interfaces. Each interrupt action function is hard coded with the peripheral instance number. Note that the IRQ file depends on the Peripheral Driver but the driver does not depend on the IRQ file.

The CMSIS startup code contains the vector table with the ISR entry names which are defined as weak references. The Peripheral Driver ISR entry names match the names defined in the vector table such that these newly-defined ISR entries in the driver replace the weak references defined in the vector table. There is no dependency on the location of the vector table. However, if the vector table is re-located (normally to RAM), the ISR entry names should remain consistent.

Each Peripheral Driver IRQ file defines the ISR entries for all instances of the peripheral instantiated on the MCU. Any unused ISR entries can be safely deleted in the user application. Note that, when building driver libraries for a particular MCU, the IRQ files are not part of the library build. When developing an application, however, include these IRQ files into the application project space.

If you want to use the HAL to directly build an interrupt-driven application or a high-level driver, define the ISR entries to service needed interrupts. The ISR entry names have to match the names of the ISR entry names provided in the CMSIS startup code vector table. Normally, the vector table is relocated to RAM during system startup and the ISR entry names remain unchanged. However, should these ISR entry names change, the user is required to register the newly defined ISR entry table names by invoking the Interrupt Manager registration function.

When working with an RTOS, if the target operating system has its own interrupt handling mechanism, the user's ISR has to dynamically register the internal vector table maintained inside the OS via the service routine provided by the OS. Some operating systems do not have dedicated interrupt handling, and no special handling is required. Most of the RTOSes designed for the Cortex-M core use PendSV exceptions for scheduling. The PendSV exceptions are low priority and execute only when all other interrupts have been serviced. There are special cases where interrupt handling prologues and epilogues have to be called at the beginning and at the end of an ISR to inform the OS of the entry and exit of an ISR so that the OS scheduler does not trigger a rescheduling event while the system is servicing interrupts. The OS Abstraction layer provides prologues and epilogues in a uniform API for all supported OS.

Peripheral Driver Types

Peripheral Drivers are designed and operated on use-case driven high-level logic. For that reason, the interaction between a Peripheral driver and other software components has a few limitations. This is a list of typical KSDK Peripheral Driver types to demonstrate this concept:

- Peripheral Drivers that use one HAL for a particular IP and system services
- Peripheral Drivers that use multiple HAL components and System Services. An example involves a Peripheral Driver that uses both the DMA HAL and the I2C HAL to build a DMA-enabled high-level I2C driver. At the same time, a DMA Peripheral Driver already exists and provides the overall DMA channel management. In this situation, issues can arise for the dedicated DMA Peripheral Driver because it is unaware of use of resources for the DMA-enabled I2C driver that accesses the DMA HAL directly. In this case, the KSDK Peripheral Drivers will only use the DMA HAL when configuring the DMA Transfer Control Descriptor for a particular channel that is dedicated to the driver and avoid using the DMA HAL to change the DMA peripheral control registers. The DMA channel allocation for a particular Peripheral Driver is normally assigned during the initialization of the peripheral by calling the DMA Peripheral Driver channel request function. In this way, the DMA maintains a used channel registry.
- Peripheral Drivers that use a combination of HAL, other Peripheral Drivers, and System Services for solution-based composite high level drivers.

Design guidelines

This section summarizes the design guidelines to develop the Peripheral Drivers. In this list, the term Peripheral Driver is replaced with the acronym “PD”:

- The PD does not directly access hardware registers and only uses HAL to interface with hardware.

It uses other PDs indirectly to access hardware functions.

- The PD does not dynamically allocate memory and does not assume any static configuration at compile-time. It supports run-time configurations by taking in a configuration data structure from the Application in an initialization function. This initialization function also takes in the memory allocated for internal operational context storage needs. The PD only reads from the configuration data structure. The configuration data structure is defined as “const”.
- The PD supports an initialization function and has a corresponding de-initialization function.
- The PD supports both blocking and non-blocking functions for data transactions.
- The PD does not depend on any software or hardware entities that do not relate to any of the peripherals available on the Kinetis platform. For example, the Ethernet (ENET) PD does not depend on an external ENET PHY, even if the ENET PD does not function without an associated external PHY.
- If the PD is built on top of HAL, the PD can work with any instance of the peripheral by taking the instance number as a parameter. When global data is shared across multiple instances, the global data access has to be protected when working with an RTOS. This can be done using the critical section functions implemented as part of the RTOS abstraction. This data access protection is addressed with the bare-metal OS abstraction implementation when no RTOS is used.
- When the ISR and other public API functions within the driver access the PD internal operation context, that data must be defined as volatile so that compiler does not inadvertently optimize the code resulting in an undesired functionality.
- The PD provides enough functions to allow the APIs provided by PD to meet the target use-case implementation. The user application should not mix the use of both a PD and HAL .
- The PD does not configure pin muxing or set up interrupt priorities for related interrupts. Those items are handled by the board-specific configuration and the application-specific logic.

Demo Applications

The Demo Applications provided in the KSDK provide examples, which show how to build user applications using the KSDK framework. The Demo Applications can be found in the KSDK top-level demo directory. The KSDK includes two types of demo applications:

- Demo Applications that demonstrate the usage of the Peripheral Drivers.
- Demo Applications that provide a reference design based on the features available on the target Kinetis MCU and evaluation boards. This type of demo is targeted to highlight a certain feature of the SOC for its intended usage, and/or to provide turnkey references using the KSDK driver library with other integrated software components such as USB stacks.

Board Configuration

The KSDK drivers make no assumption on board-specific configurations nor do the drivers configure pin muxing, which are part of the board-specific configuration. The KSDK provides board-configuration files that un-gate clocks for related I/O ports, configure pin muxing for the entire board, and functions that can be called before driver initialization. These board-configuration files can be found in the KSDK top-level board directory.

In addition, the KSDK also contains a set of drivers in the boards/common directory for common devices (such as SPI flash and ENET PHY) found on the Kinetis platform evaluation boards. These drivers are included mainly for a convenient out-of-box experience and to demonstrate how to build use-case-driven

reference designs.

OS Abstraction layer

The KSDK drivers are designed to work with or without an RTOS. The Operating System Abstraction layer (OSA) is designed and implemented for supported RTOS to provide a common set of service routines for drivers, integrated software solution, and upper-level applications so that a common code base can be used regardless the target OS.

The services supported by the OSA are driven by the driver requirements, supported middleware, and applications. The OSA layer is designed to be as thin as possible. The OSA either maps the desired services to the services provided by the target OS or implements the services that the target OS does not support.

The OSA itself does not dynamically allocate memory for implementing an OS service component. Instead, the memory for OS components is allocated by the caller and is passed into the OSA for servicing. The OSA drivers can be found in the KSDK platform/osa directory.

Software Stacks and other Middleware integration

The core of the Kinetis SDK is a set of common driver libraries built for all Kinetis MCU product family peripherals. Kinetis SDK also provides a foundation for software stacks and other middleware. The KSD-K integrates other Freescale or third party enablement software stacks, such as a USB stack and a TCP/IP stack and other middleware, to offer a complete, easy-to-use, software development kit to the Kinetis MCU users. The KSDK framework integrates all Kinetis MCU software solutions for the Kinetis product families.

Chapter 3

Analog-to-Digital Converter (ADC)

The Kinetis SDK provides both HAL and Peripheral drivers for the Analog-to-Digital Converter (ADC) block of Kinetis devices.

Modules

- [ADC HAL Driver](#)

This part describes the programming interface of the ADC HAL driver.

- [ADC Peripheral Driver](#)

This part describes the programming interface of the ADC Peripheral driver.

3.1 ADC HAL Driver

This chapter describes the programming interface of the ADC HAL driver.

Enumerations

- enum `adc_status_t` {
 `kStatus_ADC_Success` = 0U,
 `kStatus_ADC_InvalidArgument` = 1U,
 `kStatus_ADC_Failed` = 2U }
ADC status return codes.
- enum `adc_clk_divider_mode_t` {
 `kAdcClkDividerInputOf1` = 0U,
 `kAdcClkDividerInputOf2` = 1U,
 `kAdcClkDividerInputOf4` = 2U,
 `kAdcClkDividerInputOf8` = 3U }
Defines the type of the enumerating divider for the converter.
- enum `adc_resolution_mode_t` {
 `kAdcResolutionBitOf8or9` = 0U,
 `kAdcResolutionBitOfSingleEndAs8` = `kAdcResolutionBitOf8or9`,
 `kAdcResolutionBitOfDiffModeAs9` = `kAdcResolutionBitOf8or9`,
 `kAdcResolutionBitOf12or13` = 1U,
 `kAdcResolutionBitOfSingleEndAs12` = `kAdcResolutionBitOf12or13`,
 `kAdcResolutionBitOfDiffModeAs13` = `kAdcResolutionBitOf12or13`,
 `kAdcResolutionBitOf10or11` = 2U,
 `kAdcResolutionBitOfSingleEndAs10` = `kAdcResolutionBitOf10or11`,
 `kAdcResolutionBitOfDiffModeAs11` = `kAdcResolutionBitOf10or11` }
Defines the type of the enumerating resolution for the converter.
- enum `adc_clk_src_mode_t` {
 `kAdcClkSrcOfBusClk` = 0U,
 `kAdcClkSrcOfBusOrAltClk2` = 1U,
 `kAdcClkSrcOfAltClk` = 2U,
 `kAdcClkSrcOfAsynClk` = 3U }
Defines the type of the enumerating source of the input clock.
- enum `adc_long_sample_cycle_mode_t` {
 `kAdcLongSampleCycleOf24` = 0U,
 `kAdcLongSampleCycleOf16` = 1U,
 `kAdcLongSampleCycleOf10` = 2U,
 `kAdcLongSampleCycleOf4` = 3U }
- enum `adc_ref_volt_src_mode_t` {
 `kAdcRefVoltSrcOfVref` = 0U,
 `kAdcRefVoltSrcOfValt` = 1U }
- enum `adc_hw_cmp_range_mode_t` {
 `kAdcHwCmpRangeModeOf1` = 0U,
 `kAdcHwCmpRangeModeOf2` = 1U,
 `kAdcHwCmpRangeModeOf3` = 2U,

```
kAdcHwCmpRangeModeOf4 = 3U }
```

Defines the type of the enumerating asserted range in the hardware compare.

Functions

- void [ADC_HAL_Init](#) (uint32_t baseAddr)

Resets all registers into a known state for the ADC module.
- static void [ADC_HAL_ConfigChn](#) (uint32_t baseAddr, uint32_t chnGroup, bool intEnable, bool diffEnable, uint8_t chnNum)

Configures the conversion channel for the ADC module.
- static bool [ADC_HAL_GetChnConvCompletedCmd](#) (uint32_t baseAddr, uint32_t chnGroup)

Checks whether the channel conversion is completed.
- static void [ADC_HAL_SetLowPowerCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the low power mode for ADC module.
- static void [ADC_HAL_SetClkDividerMode](#) (uint32_t baseAddr, [adc_clk_divider_mode_t](#) mode)

Selects the clock divider mode for the ADC module.
- static void [ADC_HAL_SetLongSampleCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the long sample mode for the ADC module.
- static void [ADC_HAL_SetResolutionMode](#) (uint32_t baseAddr, [adc_resolution_mode_t](#) mode)

Selects the conversion resolution mode for ADC module.
- static [adc_resolution_mode_t](#) [ADC_HAL_GetResolutionMode](#) (uint32_t baseAddr)

Gets the conversion resolution mode for ADC module.
- static void [ADC_HAL_SetClkSrcMode](#) (uint32_t baseAddr, [adc_clk_src_mode_t](#) mode)

Selects the input clock source for the ADC module.
- static void [ADC_HAL_SetAsyncClkCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the asynchronous clock for the ADC module.
- static void [ADC_HAL_SetHighSpeedCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the high speed mode for the ADC module.
- static void [ADC_HAL_SetLongSampleCycleMode](#) (uint32_t baseAddr, [adc_long_sample_cycle_mode_t](#) mode)

Selects the long sample cycle mode for the ADC module.
- static uint16_t [ADC_HAL_GetChnConvValueRAW](#) (uint32_t baseAddr, uint32_t chnGroup)

Gets the raw result data of channel conversion for the ADC module.
- static void [ADC_HAL_SetHwCmpValue1](#) (uint32_t baseAddr, uint16_t value)

Sets the compare value of the lower limitation for the ADC module.
- static void [ADC_HAL_SetHwCmpValue2](#) (uint32_t baseAddr, uint16_t value)

Sets the compare value of the higher limitation for the ADC module.
- static bool [ADC_HAL_GetConvActiveCmd](#) (uint32_t baseAddr)

Checks whether the converter is active for the ADC module.
- static void [ADC_HAL_SetHwTriggerCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the hardware trigger mode for the ADC module.
- static void [ADC_HAL_SetHwCmpCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the hardware comparator for the ADC module.
- static void [ADC_HAL_SetHwCmpGreaterCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the setting that is greater than the hardware comparator.
- static void [ADC_HAL_SetHwCmpRangeCmd](#) (uint32_t baseAddr, bool enable)

Switches to enable the setting of the range for hardware comparator.
- void [ADC_HAL_SetHwCmpMode](#) (uint32_t baseAddr, [adc_hw_cmp_range_mode_t](#) mode)

Configures the asserted range of the hardware comparator for the ADC module.
- static void [ADC_HAL_SetRefVoltSrcMode](#) (uint32_t baseAddr, [adc_ref_volt_src_mode_t](#) mode)

ADC HAL Driver

Selects the reference voltage source for the ADC module.

- static void [ADC_HAL_SetContinuousConvCmd](#) (uint32_t baseAddr, bool enable)
Switches to enable the continuous conversion mode for the ADC module.

3.1.0.1 ADC HAL Driver

Overview

This chapter describes the programming interface of the ADC HAL driver.

3.1.1 Enumeration Type Documentation

3.1.1.1 enum adc_status_t

Enumerator

kStatus_ADC_Success Success.

kStatus_ADC_InvalidArgument Invalid argument existed.

kStatus_ADC_Failed Execution failed.

3.1.1.2 enum adc_clk_divider_mode_t

Enumerator

kAdcClkDividerInputOf1 For divider 1 from the input clock to ADC.

kAdcClkDividerInputOf2 For divider 2 from the input clock to ADC.

kAdcClkDividerInputOf4 For divider 4 from the input clock to ADC.

kAdcClkDividerInputOf8 For divider 8 from the input clock to ADC.

3.1.1.3 enum adc_resolution_mode_t

Enumerator

kAdcResolutionBitOf8or9 8-bit for single end sample, or 9-bit for differential sample.

kAdcResolutionBitOfSingleEndAs8 8-bit for single end sample.

kAdcResolutionBitOfDiffModeAs9 9-bit for differential sample.

kAdcResolutionBitOf12or13 12-bit for single end sample, or 13-bit for differential sample.

kAdcResolutionBitOfSingleEndAs12 12-bit for single end sample.

kAdcResolutionBitOfDiffModeAs13 13-bit for differential sample.

kAdcResolutionBitOf10or11 10-bit for single end sample, or 11-bit for differential sample.

kAdcResolutionBitOfSingleEndAs10 10-bit for single end sample.

kAdcResolutionBitOfDiffModeAs11 11-bit for differential sample.

3.1.1.4 enum adc_clk_src_mode_t

Enumerator

- kAdcClkSrcOfBusClk* For input as bus clock.
- kAdcClkSrcOfBusOrAltClk2* For input as bus clock /2 or AltClk2.
- kAdcClkSrcOfAltClk* For input as alternate clock (ALTCLK).
- kAdcClkSrcOfAsynClk* For input as asynchronous clock (ADACK).

3.1.1.5 enum adc_long_sample_cycle_mode_t

Enumerator

- kAdcLongSampleCycleOf24* 20 extra ADCK cycles, 24 ADCK cycles total.
- kAdcLongSampleCycleOf16* 12 extra ADCK cycles, 16 ADCK cycles total.
- kAdcLongSampleCycleOf10* 6 extra ADCK cycles, 10 ADCK cycles total.
- kAdcLongSampleCycleOf4* 2 extra ADCK cycles, 6 ADCK cycles total.

3.1.1.6 enum adc_ref_volt_src_mode_t

Enumerator

- kAdcRefVoltSrcOfVref* For external pins pair of VrefH and VrefL.
- kAdcRefVoltSrcOfValt* For alternate reference pair of ValtH and ValtL.

3.1.1.7 enum adc_hw_cmp_range_mode_t

When the internal CMP is enabled, the COCO flag, which represents the complement of the conversion, is not asserted if the sample value is not in the indicated range. Eventually, the data of conversion result is not kept in the result data register. The two values, cmpValue1 and cmpValue2, mark the thresholds with the comparator feature. kAdcHwCmpRangeModeOf1: Both greater than and in range switchers are disabled. The available range is " $<$ cmpValue1". kAdcHwCmpRangeModeOf2: Greater than switcher is enabled while the in range switcher is disabled. The available range is " $>$ cmpValue1". kAdcHwCmpRangeModeOf3: Greater than switcher is disabled while in range switcher is enabled. The available range is " $<$ cmpValue1" or " $>$ cmpValue2" when cmpValue1 \leq cmpValue2, or " $<$ cmpValue1" and " $>$ cmpValue2" when cmpValue1 \geq cmpValue2. kAdcHwCmpRangeModeOf4: Both greater than and in range switchers are enabled. The available range is " $>$ cmpValue1" and " $<$ cmpValue2" when cmpValue1 \leq cmpValue2, or " $>$ cmpValue1" or " $<$ cmpValue2" when cmpValue1 $<$ cmpValue2.

Enumerator

- kAdcHwCmpRangeModeOf1* For selection mode 1.
- kAdcHwCmpRangeModeOf2* For selection mode 2.
- kAdcHwCmpRangeModeOf3* For selection mode 3.
- kAdcHwCmpRangeModeOf4* For selection mode 4.

3.1.2 Function Documentation

3.1.2.1 void ADC_HAL_Init (*uint32_t baseAddr*)

This function resets all registers into a known state for the ADC module. This known state is the reset value indicated by the Reference manual. It is strongly recommended to call this API before any other operation when initializing the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

3.1.2.2 static void ADC_HAL_ConfigChn (*uint32_t baseAddr, uint32_t chnGroup, bool intEnable, bool diffEnable, uint8_t chnNum*) [inline], [static]

This function configures the channel for the ADC module. At any point, only one of the configuration groups takes effect. The other channel mux of the first group (group A, 0) is only for the hardware trigger. Both software and hardware trigger can be used to the first group. When in software trigger mode, once the available channel is set, the conversion begins to execute.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>chnGroup</i>	Channel configuration group ID.
<i>intEnable</i>	Switcher to enable interrupt when conversion is completed.
<i>diffEnable</i>	Switcher to enable differential channel mode.
<i>chnNum</i>	ADC channel for next conversion.

3.1.2.3 static bool ADC_HAL_GetChnConvCompletedCmd (*uint32_t baseAddr, uint32_t chnGroup*) [inline], [static]

This function checks whether the channel conversion for the ADC module is completed.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

<i>chnGroup</i>	Channel configuration group ID.
-----------------	---------------------------------

Returns

Assertion of completed conversion mode.

3.1.2.4 static void ADC_HAL_SetLowPowerCmd (*uint32_t baseAddr*, *bool enable*) [inline], [static]

This function switches to enable the low power mode for ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.5 static void ADC_HAL_SetClkDividerMode (*uint32_t baseAddr*, *adc_clk_divider_mode_t mode*) [inline], [static]

This function selects the clock divider mode for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Selection of mode enumeration. See to "adc_clk_divider_mode_t".

3.1.2.6 static void ADC_HAL_SetLongSampleCmd (*uint32_t baseAddr*, *bool enable*) [inline], [static]

This function switches to enable the long sample mode for the ADC module. This function adjusts the sample period to allow the higher impedance inputs to be accurately sampled or to maximize the conversion speed for the lower impedance inputs. Longer sample times can also be used to lower overall power consumption if the continuous conversions are enabled and the high conversion rates are not required. If the long sample mode is enabled, more configuration is set by calling the "ADC_HAL_SetLongSampleCycleMode()" function.

ADC HAL Driver

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.7 static void ADC_HAL_SetResolutionMode (uint32_t *baseAddr*, adc_resolution_mode_t *mode*) [inline], [static]

This function selects the conversion resolution mode for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Selection of mode enumeration. See to "adc_resolution_mode_t".

3.1.2.8 static adc_resolution_mode_t ADC_HAL_GetResolutionMode (uint32_t *baseAddr*) [inline], [static]

This function gets the conversion resolution mode for the ADC module. It is specially used when processing the conversion result of RAW format.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

Current conversion resolution mode.

3.1.2.9 static void ADC_HAL_SetClkSrcMode (uint32_t *baseAddr*, adc_clk_src_mode_t *mode*) [inline], [static]

This function selects the input clock source for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Selection of mode enumeration. See to "adc_clk_src_mode_t".

3.1.2.10 static void ADC_HAL_SetAsyncClkCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the asynchronous clock for the ADC module. It enables the ADC's asynchronous clock source and the clock source output regardless of the conversion and the input clock select status of the ADC. Asserting this function allows the clock to be used even while the ADC is idle or operating from a different clock source. Also, latency of initiating a single or first-continuous conversion with the asynchronous clock selected is reduced since the ADC internal clock has been already operational.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.11 static void ADC_HAL_SetHighSpeedCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the high speed mode for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.12 static void ADC_HAL_SetLongSampleCycleMode (uint32_t *baseAddr*, adc_long_sample_cycle_mode_t *mode*) [inline], [static]

This function selects the long sample cycle mode for the ADC module. This function should be called along with "ADC_HAL_SetLongSampleCmd()".

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

ADC HAL Driver

<i>mode</i>	Selection of long sample cycle mode. See the "adc_long_sample_cycle_mode_t".
-------------	--

3.1.2.13 static uint16_t ADC_HAL_GetChnConvValueRAW (uint32_t *baseAddr*, uint32_t *chnGroup*) [inline], [static]

This function gets the result data of conversion for the ADC module. The return value is raw data that is not processed. The unavailable bits would be filled with "0" in single-ended mode and sign bit in differential mode.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>chnGroup</i>	Channel configuration group ID.

Returns

Conversion value of RAW.

3.1.2.14 static void ADC_HAL_SetHwCmpValue1 (uint32_t *baseAddr*, uint16_t *value*) [inline], [static]

This function sets the compare value of the lower limitation for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>value</i>	Setting value.

3.1.2.15 static void ADC_HAL_SetHwCmpValue2 (uint32_t *baseAddr*, uint16_t *value*) [inline], [static]

This function sets the compare value of the higher limitation for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

<i>value</i>	Setting value.
--------------	----------------

3.1.2.16 static bool ADC_HAL_GetConvActiveCmd (*uint32_t baseAddr*) [inline], [static]

This function checks whether the converter is active for the ADC module. If it is dis-asserted when the conversion is completed, one of the completed flag is asserted for the indicated group mux. See the "ADC_HAL_GetChnConvCompletedCmd()".

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

Assertion of that the converter is active.

3.1.2.17 static void ADC_HAL_SetHwTriggerCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the hardware trigger mode for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.18 static void ADC_HAL_SetHwCmpCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the hardware comparator for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

ADC HAL Driver

<i>enable</i>	Switcher to asserted the feature.
---------------	-----------------------------------

3.1.2.19 static void ADC_HAL_SetHwCmpGreaterCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the setting that is greater than the hardware comparator.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.20 static void ADC_HAL_SetHwCmpRangeCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the setting of range for the hardware comparator.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

3.1.2.21 void ADC_HAL_SetHwCmpMode (*uint32_t baseAddr, adc_hw_cmp_range_mode_t mode*)

This function configures the asserted range of the hardware comparator for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Selection of range mode, see to "adc_hw_cmp_range_mode_t".

3.1.2.22 static void ADC_HAL_SetRefVoltSrcMode (*uint32_t baseAddr, adc_ref_volt_src_mode_t mode*) [inline], [static]

This function selects the reference voltage source for the ADC module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Selection of asserted the feature.

3.1.2.23 static void ADC_HAL_SetContinuousConvCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the continuous conversion mode for the ADC module. Once enabled, continuous conversions, or sets of conversions if the hardware average function, is enabled after initiating a conversion.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to asserted the feature.

ADC Peripheral Driver

3.2 ADC Peripheral Driver

This chapter describes the programming interface of the ADC Peripheral driver.

Data Structures

- struct [adc_hw_cmp_config_t](#)
Defines the structure to configure the ADC module hardware compare. [More...](#)
- struct [adc_chn_config_t](#)
Defines the structure to configure the ADC channel. [More...](#)
- struct [adc_user_config_t](#)
Defines the structure to initialize the ADC module converter. [More...](#)
- struct [adc_state_t](#)
Internal driver state information. [More...](#)

Typedefs

- typedef void(* [adc_callback_t](#))(void)
Defines the type of a user-defined callback function.

Functions

- [adc_status_t ADC_DRV_StructInitUserConfigForIntMode](#) ([adc_user_config_t](#) *userConfigPtr)
Fills the initial user configuration for the interrupt mode.
- [adc_status_t ADC_DRV_StructInitUserConfigForBlockingMode](#) ([adc_user_config_t](#) *userConfigPtr)
Fills the initial user configuration for blocking mode.
- [adc_status_t ADC_DRV_StructInitUserConfigForOneTimeTriggerMode](#) ([adc_user_config_t](#) *userConfigPtr)
Fills the initial user configuration for a one-time trigger mode.
- [adc_status_t ADC_DRV_Init](#) (uint32_t instance, [adc_user_config_t](#) *userConfigPtr, [adc_state_t](#) *userStatePtr)
Initializes the ADC module converter.
- [void ADC_DRV_Deinit](#) (uint32_t instance)
De-initializes the ADC module converter.
- [void ADC_DRV_EnableLongSample](#) (uint32_t instance, [adc_long_sample_cycle_mode_t](#) mode)
Enables the long sample mode feature.
- [void ADC_DRV_DisableLongSample](#) (uint32_t instance)
Disables the long sample mode feature.
- [adc_status_t ADC_DRV_EnableHwCmp](#) (uint32_t instance, [adc_hw_cmp_config_t](#) *hwCmpConfigPtr)
Enables the hardware compare feature.
- [void ADC_DRV_DisableHwCmp](#) (uint32_t instance)
Disables the hardware compare feature.
- [adc_status_t ADC_DRV_ConfigConvChn](#) (uint32_t instance, uint32_t chnGroup, [adc_chn_config_t](#) *chnConfigPtr)
Configures the conversion channel by software.

- void [ADC_DRV_WaitConvDone](#) (uint32_t instance, uint32_t chnGroup)
Waits for the latest conversion to be complete.
- void [ADC_DRV_PauseConv](#) (uint32_t instance, uint32_t chnGroup)
Pauses the current conversion by software.
- uint16_t [ADC_DRV_GetConvValueRAW](#) (uint32_t instance, uint32_t chnGroup)
Gets the latest conversion value.
- uint16_t [ADC_DRV_GetConvValueRAWInt](#) (uint32_t instance, uint32_t chnGroup)
Gets the latest conversion value in the buffer when using the interrupt mode.
- int32_t [ADC_DRV_ConvRAWData](#) (uint16_t convValue, bool diffEnable, [adc_resolution_mode_t](#) mode)
Formats the initial value fetched from the ADC module.
- [adc_status_t ADC_DRV_InstallCallback](#) (uint32_t instance, uint32_t chnGroup, [adc_callback_t](#) userCallback)
Installs the user-defined callback in the ADC module.
- void [ADC_DRV_IRQHandler](#) (uint32_t instance)
Driver-defined ISR in the ADC module.

3.2.0.24 ADC Peripheral Driver

Overview

The ADC peripheral driver configures the ADC (Analog-to-Digital Converter). It handles calibration, initialization and configuration of ADC module.

Driver model building

ADC driver has three parts:

- Basic Converter - This part handles the mechanism that converts the external analog voltage to digital value. The API functions configure the converter.
- Channel Mux - Multiple channels share the converter in each ADC instance because of the time division multiplexing. However, the converter can only handle one channel at a time. To get the value of an indicated channel, the channel mux should be set to the connection between an indicated pad and the converter's input. The conversion value during this period is for the channel only. The API functions configure the channel.
- Advance Feature Group - The advanced feature group covers optional features for applications. These features includes some that are already implemented by hardware, such as the calibration, hardware average, hardware compare, different power, and speed mode. The APIs for this part configure the advanced features. Although these features are optional, they are recommended to ensure that the ADC performs better, especially for calibration.

Initialization

Note that the calibration should be done before all the other operations.

ADC Peripheral Driver

To initialize the ADC driver, a configuration structure is needed and should be filled with an available configuration. To make it easier to fill the structure, API functions have been designed for typical use cases. See the "Call diagram" chapter for typical use cases. Additionally, the application should provide a block of memory to keep the state while the driver operates. After the configuration structure is available and memory is allocated to keep state, the ADC module can be initialized by calling the API of [ADC_DRV_Init\(\)](#) function.

Call diagram

Three kinds of typical use cases are designed for the ADC module:

- Interrupt mode - Interrupt mode works alone with continuous conversion. If it is chosen, the internal ADC ISR, inside the ADC PD driver, moves the conversion value from the result register to the internal buffer after the conversion is complete. As a result, the ADC data buffer is always updated continuously. The API function, which is reading the buffer, returns the latest conversion result to the application. Use the interrupt mode carefully with the continuous conversion, because too many interrupts might affect the main routine. You should use the low conversion speed in this mode.
- Blocking mode. Blocking mode is working alone with continuous conversion mode too, but not enabling the interrupt. After trigger by configuring the channel, the conversion is launched. When conversion is completed, it would be blocked since the result data is still not read. It is so called blocking mode. Application should fetch the result data by API, and then the conversion could be continuous.
- One-Time-Trigger Mode. One-Time-Trigger works without interrupt and continuous mode. Once triggered by configuring the channel, the conversion launches and the application fetches the result data. There is no auto operation to make the conversion continuous. The application should trigger the conversion again if another conversion is needed.

The three use cases are all based on the software trigger. However, they can easily be ported to use the hardware trigger. The only modification is to enable the hardware trigger when initializing the converter.

The complex use cases, such as the DMA and the multiple channel scan need another module to work correctly. They can be customized according to the application.

These are the examples to initialize and configure the ADC driver for typical use cases.

Interrupt Mode:

```
/* adc_test_int.c */

#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include "fsl_adc_driver.h"
#include "fsl_os_abstraction.h"

static uint32_t MyIsrCounter = 0U;
void ADC_TEST_MyIsr(void);

void ADC_TEST_IntMode(uint32_t instance, uint8_t chn, uint32_t times)
{
#if FSL_FEATURE_ADC_HAS_CALIBRATION
    adc_calibration_param_t MyAdcCalibraitionParam;
```

```

#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */
adc_user_config_t MyAdcUserConfig;
adc_state_t MyAdcState;
adc_chn_config_t MyChnConfig;
volatile int32_t MyAdcValue;
uint32_t LoopCounter = 0U;

#if FSL_FEATURE_ADC_HAS_CALIBRATION
/* Auto calibration. */
ADC_DRV_GetAutoCalibrationParam(instance, &MyAdcCalibraitionParam);
ADC_DRV_SetCalibrationParam(instance, &MyAdcCalibraitionParam);
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */

/* Initialization for interrupt mode. */
ADC_DRV_StructInitUserConfigForIntMode(&MyAdcUserConfig);
ADC_DRV_Init(instance, &MyAdcUserConfig, &MyAdcState);

/* Install Callback function into ISR. */
ADC_DRV_InstallCallback(instance, 0U, ADC_TEST_MyIsr);
/* Trigger indicated channel. */
MyChnConfig.chnNum = chn;
MyChnConfig.diffEnable = false;
MyChnConfig.intEnable = true;
MyChnConfig.chnMux = kAdcChnMuxOfDefault;
ADC_DRV_ConfigConvChn(instance, 0U, &MyChnConfig);

while (LoopCounter < times)
{
    /* Fetch the conversion value and format it. */
    MyAdcValue = ADC_DRV_GetConvValueRAWInt(instance, 0U);
    printf("ADC_DRV_GetConvValueRAWInt: %d\r\n", MyAdcValue);
    MyAdcValue = ADC_DRV_ConvRAWData(MyAdcValue, false,
        kAdcResolutionBitOfSingleEndAs12);
    printf("ADC_DRV_ConvRAWData: %d\r\n", MyAdcValue);
    /* To do something. */
    LoopCounter++;
    OSA_TimeDelay(200);
}
/* Pause the conversion after testing. */
ADC_DRV_PauseConv(instance, 0U);
/* Disable the ADC. */
ADC_DRV_Deinit(instance);
}

void ADC_TEST_MyIsr(void)
{
    MyIsrCounter++;
}

```

Blocking Mode:

```

/* adc_test_blocking.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_adc_driver.h"
#include "fsl_os_abstraction.h"

void ADC_TEST_Blocking(uint32_t instance, uint8_t chn, uint32_t times)
{
#if FSL_FEATURE_ADC_HAS_CALIBRATION
    adc_calibration_param_t MyAdcCalibraitionParam;
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */
    adc_user_config_t MyAdcUserConfig;
    adc_state_t MyAdcState;

```

ADC Peripheral Driver

```
adc_chn_config_t MyChnConfig;
volatile int32_t MyAdcValue;
uint32_t LoopCounter = 0U;

#if FSL_FEATURE_ADC_HAS_CALIBRATION
/* Auto calibraion. */
ADC_DRV_SetAutoCalibrationParam(instance, &MyAdcCalibraitionParam);
ADC_DRV_SetCalibrationParam(instance, &MyAdcCalibraitionParam);
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */

/* Initialization for interrupt mode. */
ADC_DRV_StructInitUserConfigForBlockingMode(&MyAdcUserConfig
);
ADC_DRV_Init(instance, &MyAdcUserConfig, &MyAdcState);

/* Trigger the channel 0. */
MyChnConfig.chnNum = chn;
MyChnConfig.diffEnable= false;
MyChnConfig.intEnable = false;
MyChnConfig.chnMux = kAdcChnMuxOfDefault;
ADC_DRV_ConfigConvChn(instance, 0U, &MyChnConfig);

while (LoopCounter < times)
{
    ADC_DRV_WaitConvDone(instance, 0U);

    /* Fetch the conversion value and format it. */
    MyAdcValue = ADC_DRV_GetConvValueRAW(instance, 0U);
    printf("ADC_DRV_GetConvValueRAW: %d\r\n", MyAdcValue);
    MyAdcValue = ADC_DRV_ConvRAWData(MyAdcValue, false,
kAdcResolutionBitOfSingleEndAs12);
    printf("ADC_DRV_ConvRAWData: %d\r\n", MyAdcValue);

    /* To do something. */
    LoopCounter++;
    OSA_TimeDelay(200);
}
/* Pause the conversion after testing. */
ADC_DRV_PauseConv(instance, 0U);
/* Disable the ADC. */
ADC_DRV_Deinit(instance);
}
```

One-Time-Trigger Mode:

```
/* adc_test_one_time_trigger.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_adc_driver.h"
#include "fsl_os_abstraction.h"

void ADC_TEST_OneTimeTrigger(uint32_t instance, uint8_t chn, uint32_t times)
{
#if FSL_FEATURE_ADC_HAS_CALIBRATION
    adc_calibration_param_t MyAdcCalibraitionParam;
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */
    adc_user_config_t MyAdcUserConfig;
    adc_state_t MyAdcState;
    adc_chn_config_t MyChnConfig;
    volatile int32_t MyAdcValue;
    uint32_t LoopCounter = 0U;

#if FSL_FEATURE_ADC_HAS_CALIBRATION
/* Auto calibraion. */
```

```

ADC_DRV_GetAutoCalibrationParam(instance, &MyAdcCalibraitionParam);
ADC_DRV_SetCalibrationParam(instance, &MyAdcCalibraitionParam);
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */

/* Initialization for interrupt mode. */
ADC_DRV_StructInitUserConfigForOneTimeTriggerMode(&
    MyAdcUserConfig);
ADC_DRV_Init(instance, &MyAdcUserConfig, &MyAdcState);

/* Setting for channel 0. */
MyChnConfig.chnNum = chn;
MyChnConfig.diffEnable= false;
MyChnConfig.intEnable = false;
MyChnConfig.chnMux = kAdcChnMuxOfDefault;

while (LoopCounter < times)
{
    /* Trigger the conversion with indicated channel's configuration. */
    ADC_DRV_ConfigConvChn(instance, 0U, &MyChnConfig);
    /* Wait for the conversion to be done. */
    ADC_DRV_WaitConvDone(instance, 0U);
    /* Fetch the conversion value and format it. */
    MyAdcValue = ADC_DRV_GetConvValueRAW(instance, 0U);
    printf("ADC_DRV_GetConvValueRAW: %d\r\n", MyAdcValue);
    MyAdcValue = ADC_DRV_ConvRAWData(MyAdcValue, false,
        kAdcResolutionBitOfSingleEndAs12);
    printf("ADC_DRV_ConvRAWData: %d\r\n", MyAdcValue);

    /* To do something. */
    LoopCounter++;
    OSA_TimeDelay(200);
}
/* Pause the conversion after testing. */
ADC_DRV_PauseConv(instance, 0U);
/* Disable the ADC. */
ADC_DRV_Deinit(instance);
}

```

For Advanced Features:

```

/* adc_test_advanced_feature.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_adc_driver.h"
#include "fsl_os_abstraction.h"

void ADC_TEST_AdvancedFeature(uint32_t instance, uint8_t chn, uint32_t times)
{
#if FSL_FEATURE_ADC_HAS_CALIBRATION
    adc_calibration_param_t MyAdcCalibraitionParam;
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */
    adc_user_config_t MyAdcUserConfig;
    adc_state_t MyAdcState;
    adc_chn_config_t MyChnConfig;
    volatile int32_t MyAdcValue;
    uint32_t LoopCounter = 0U;
    adc_hw_cmp_config_t MyAdcHwCmpConfig;

#if FSL_FEATURE_ADC_HAS_CALIBRATION
    /* Auto calibraion. */
    ADC_DRV_GetAutoCalibrationParam(instance, &MyAdcCalibraitionParam);
    ADC_DRV_SetCalibrationParam(instance, &MyAdcCalibraitionParam);
#endif /* FSL_FEATURE_ADC_HAS_CALIBRATION */
}

```

ADC Peripheral Driver

```
/* Initialization for interrupt mode. */
ADC_DRV_SetupUserConfigForOneTimeTriggerMode(&
    MyAdcUserConfig);
ADC_DRV_Init(instance, &MyAdcUserConfig, &MyAdcState);

/* Setting for advanced feature. */
/* Hardware compare. */
MyAdcHwCmpConfig.cmpRangeMode = kAdcHwCmpRangeModeOf4;
    /* Available range is between cmpValue1 and cmpValue2. */
MyAdcHwCmpConfig.cmpValue1 = 0x0010U;
MyAdcHwCmpConfig.cmpValue2 = 0xFFFF0U;
ADC_DRV_EnableHwCmp(instance, &MyAdcHwCmpConfig);
/* Long sample mode. */
ADC_DRV_EnableLongSample(instance,
    kAdcLongSampleCycleOf16);
/* Hardware average. */
#if FSL_FEATURE_ADC_HAS_HW_AVERAGE
    ADC_DRV_EnableHwAverage(instance, kAdcHwAverageCountOf32);
#endif /* FSL_FEATURE_ADC_HAS_HW_AVERAGE */

/* Setting for channel. */
MyChnConfig.chnNum = chn;
MyChnConfig.diffEnable= false;
MyChnConfig.intEnable = false;
MyChnConfig.chnMux = kAdcChnMuxOfDefault;

while (LoopCounter < times)
{
    /* Trigger the conversion with indicated channel's configuration. */
    ADC_DRV_ConfigConvChn(instance, 0U, &MyChnConfig);
    /* Wait for the conversion to be done. */
    ADC_DRV_WaitConvDone(instance, 0U);
    /* Fetch the conversion value and format it. */
    MyAdcValue = ADC_DRV_GetConvValueRAW(instance, 0U);
    printf("ADC_DRV_GetConvValueRAW: %d\r\n", MyAdcValue);
    MyAdcValue = ADC_DRV_ConvRAWData(MyAdcValue, false,
        kAdcResolutionBitOfSingleEndAs12);
    printf("ADC_DRV_ConvRAWData: %d\r\n", MyAdcValue);
    /* To do something. */
    LoopCounter++;
    OSA_TimeDelay(200);
}
/* Pause the conversion after testing. */
ADC_DRV_PauseConv(instance, 0U);
/* Disable the ADC. */
ADC_DRV_Deinit(instance);
}
```

3.2.1 Data Structure Documentation

3.2.1.1 struct adc_hw_cmp_config_t

This structure keeps the configuration for the ADC internal comparator.

Data Fields

- uint16_t cmpValue1
Value for CMP value 1.
- uint16_t cmpValue2
Value for CMP value 2.

3.2.1.1.0.1.1 Field Documentation

3.2.1.1.0.1.1 uint16_t adc_hw_cmp_config_t::cmpValue1

3.2.1.1.0.1.2 uint16_t adc_hw_cmp_config_t::cmpValue2

Select the available range to pass the comparator.

3.2.1.2 struct adc_chn_config_t

This structure keeps the ADC channel configuration.

Data Fields

- **uint32_t chnNum**
Selection of input channel number.
- **bool intEnable**
Enable trigger interrupt when the conversion is complete.
- **bool diffEnable**
Enable setting the differential mode for conversion.

3.2.1.2.0.2 Field Documentation

3.2.1.2.0.2.1 uint32_t adc_chn_config_t::chnNum

3.2.1.2.0.2.2 bool adc_chn_config_t::intEnable

3.2.1.2.0.2.3 bool adc_chn_config_t::diffEnable

3.2.1.3 struct adc_user_config_t

This structure keeps the configuration for the ADC module converter.

Data Fields

- **bool intEnable**
Enable internal ISR.
- **bool lowPowerEnable**
Enable low power mode for converter.
- **adc_clk_divider_mode_t clkDividerMode**
Select divider of input clock for converter.
- **adc_resolution_mode_t resolutionMode**
Select conversion resolution for converter.
- **adc_clk_src_mode_t clkSrcMode**
Select source of input clock for converter.
- **bool asyncClkEnable**
Enable asynchronous clock output at when initializing converter.
- **bool highSpeedEnable**

ADC Peripheral Driver

- **bool hwTriggerEnable**
Enable the high speed mode for converter.
- **adc_ref_volt_src_mode_t refVoltSrcMode**
Select the reference voltage source for converter.
- **bool continuousConvEnable**
Enable launching the continuous conversion mode.

3.2.1.3.0.3 Field Documentation

3.2.1.3.0.3.1 bool adc_user_config_t::intEnable

3.2.1.3.0.3.2 bool adc_user_config_t::lowPowerEnable

3.2.1.3.0.3.3 adc_clk_divider_mode_t adc_user_config_t::clkDividerMode

3.2.1.3.0.3.4 adc_resolution_mode_t adc_user_config_t::resolutionMode

3.2.1.3.0.3.5 adc_clk_src_mode_t adc_user_config_t::clkSrcMode

3.2.1.3.0.3.6 bool adc_user_config_t::asyncClkEnable

3.2.1.3.0.3.7 bool adc_user_config_t::highSpeedEnable

3.2.1.3.0.3.8 bool adc_user_config_t::hwTriggerEnable

3.2.1.3.0.3.9 adc_ref_volt_src_mode_t adc_user_config_t::refVoltSrcMode

3.2.1.3.0.3.10 bool adc_user_config_t::continuousConvEnable

3.2.1.4 struct adc_state_t

The contents of this structure are internal to the driver and should not be modified by users. Also, contents of the structure are subject to change in future releases.

3.2.1.4.0.4 Field Documentation

3.2.1.4.0.4.1 adc_callback_t adc_state_t::userCallbackFunc

3.2.1.4.0.4.2 uint16_t adc_state_t::convBuff

3.2.2 Function Documentation

3.2.2.1 adc_status_t ADC_DRV_StructInitUserConfigForIntMode (adc_user_config_t * userConfigPtr)

This function fills the initial user configuration for interrupt mode. Calling the initialization function with the filled parameter configures the ADC module to function in the interrupt mode. The settings are:

.intEnable = true; .lowPowerEnable = true; .clkDividerMode = kAdcClkDividerInputOf8; .resolution-

```
Mode = kAdcResolutionBitOf12or13; .clkSrcMode = kAdcClkSrcOfAsynClk; .asyncClkEnable = true;
.highSpeedEnable = false; .hwTriggerEnable = false; .dmaEnable = false; .refVoltSrcMode = kAdcRef-
VoltSrcOfVref; .continuousConvEnable = true;
```

Parameters

<i>userConfigPtr</i>	Pointer to the user configuration structure. See the "adc_user_config_t".
----------------------	---

Returns

Execution status.

3.2.2.2 adc_status_t ADC_DRV_StructInitUserConfigForBlockingMode (adc_user_config_t * *userConfigPtr*)

This function fills the initial user configuration for blocking mode. Calling the initialization function with the filled parameter configures the ADC module to function in the blocking mode. The settings are:

```
.intEnable = false; .lowPowerEnable = false; .clkDividerMode = kAdcClkDividerInputOf8; .resolution-
Mode = kAdcResolutionBitOf12or13; .clkSrcMode = kAdcClkSrcOfBusClk; .asyncClkEnable = false;
.highSpeedEnable = false; .hwTriggerEnable = false; .dmaEnable = false; .refVoltSrcMode = kAdcRef-
VoltSrcOfVref; .continuousConvEnable = true;
```

Parameters

<i>userConfigPtr</i>	Pointer to the user configuration structure. See the "adc_user_config_t".
----------------------	---

Returns

Execution status.

3.2.2.3 adc_status_t ADC_DRV_StructInitUserConfigForOneTimeTriggerMode (adc_user_config_t * *userConfigPtr*)

This function fills the initial user configuration for a one-time trigger mode. Calling the initialization function with the filled parameter configures the ADC module to function in the one-time trigger mode. The settings are:

```
.intEnable = false; .lowPowerEnable = false; .clkDividerMode = kAdcClkDividerInputOf8; .resolution-
Mode = kAdcResolutionBitOf12or13; .clkSrcMode = kAdcClkSrcOfBusClk; .asyncClkEnable = false;
.highSpeedEnable = false; .hwTriggerEnable = false; .dmaEnable = false; .refVoltSrcMode = kAdcRef-
VoltSrcOfVref; .continuousConvEnable = false;
```

ADC Peripheral Driver

Parameters

<i>userConfigPtr</i>	Pointer to the user configuration structure. See the "adc_user_config_t".
----------------------	---

Returns

Execution status.

3.2.2.4 **adc_status_t ADC_DRV_Init (uint32_t *instance*, adc_user_config_t * *userConfigPtr*, adc_state_t * *userStatePtr*)**

This function initializes the converter in the ADC module. Regardless of the completed calibration for the device, this API function, with initial configuration should be called before any other operations. Initial configurations are mainly for the converter itself. For advanced features, the corresponding APIs should be called after this function.

Parameters

<i>instance</i>	ADC instance ID.
<i>userConfigPtr</i>	Pointer to the initialization structure. See the "adc_user_config_t".
<i>userStatePtr</i>	Pointer to the context memory structure. See the "adc_state_t".

Returns

Execution status.

3.2.2.5 **void ADC_DRV_Deinit (uint32_t *instance*)**

This function de-initializes the ADC module. It gates the ADC module. When ADC is no longer used in application, calling this API function shuts down the device to reduce the power consumption.

Parameters

<i>instance</i>	ADC instance ID.
-----------------	------------------

3.2.2.6 **void ADC_DRV_EnableLongSample (uint32_t *instance*, adc_long_sample_cycle_mode_t *mode*)**

This function enables the long sample mode feature and sets it with the indicated configuration. Launching the long sample mode adjusts the sample period to allow the higher impedance inputs to be accurately sampled.

Parameters

<i>instance</i>	ADC instance ID.
<i>mode</i>	Selection of configuration, see to "adc_long_sample_cycle_mode_t".

3.2.2.7 void ADC_DRV_DisableLongSample (uint32_t *instance*)

This API function disables the long sample mode feature and

Parameters

<i>instance</i>	ADC instance ID.
-----------------	------------------

3.2.2.8 adc_status_t ADC_DRV_EnableHwCmp (uint32_t *instance*, adc_hw_cmp_config_t * *hwCmpConfigPtr*)

This API enables the hardware compare feature with the indicated configuration. Launching the hardware compare ensures that the conversion, which results in a predefined range, can be only accepted. Values out of range are ignored during conversion.

Parameters

<i>instance</i>	ADC instance ID.
<i>hwCmpConfig- Ptr</i>	Pointer to a configuration structure. See the "adc_hw_cmp_config_t".

Returns

Execution status.

3.2.2.9 void ADC_DRV_DisableHwCmp (uint32_t *instance*)

This API function disables the hardware compare feature.

Parameters

ADC Peripheral Driver

<i>instance</i>	ADC instance ID.
-----------------	------------------

3.2.2.10 `adc_status_t ADC_DRV_ConfigConvChn (uint32_t instance, uint32_t chnGroup, adc_chn_config_t * chnConfigPtr)`

This function configures the conversion channel. When the ADC module has been initialized by enabling the software trigger (disable hardware trigger), calling this API triggers the conversion.

Parameters

<i>instance</i>	ADC instance ID.
<i>chnGroup</i>	Selection of the configuration group.
<i>chnConfigPtr</i>	Pointer to the configuration structure. See the "adc_chn_config_t".

Returns

Execution status.

3.2.2.11 `void ADC_DRV_WaitConvDone (uint32_t instance, uint32_t chnGroup)`

This function waits for the latest conversion to be complete. When triggering the conversion by configuring the available channel, the converter is launched. This API function should be called to wait for the conversion to be complete when no interrupt or DMA mode is used for the ADC module. After the waiting period, the available data from the conversion result are fetched. The complete flag is not cleared until the result data is read.

Parameters

<i>instance</i>	ADC instance ID.
<i>chnGroup</i>	Selection of configuration group.

Returns

Execution status.

3.2.2.12 `void ADC_DRV_PauseConv (uint32_t instance, uint32_t chnGroup)`

This function pauses the current conversion setting by software.

Parameters

<i>instance</i>	ADC instance ID.
<i>chnGroup</i>	Selection of configuration group.

3.2.2.13 `uint16_t ADC_DRV_GetConvValueRAW (uint32_t instance, uint32_t chnGroup)`

This function gets the conversion value from the ADC module.

Parameters

<i>instance</i>	ADC instance ID.
<i>chnGroup</i>	Selection of configuration group.

Returns

Unformatted conversion value.

3.2.2.14 `uint16_t ADC_DRV_GetConvValueRAWInt (uint32_t instance, uint32_t chnGroup)`

This function gets the latest conversion value in the buffer when using the interrupt mode. Note that this function is only available in the interrupt mode.

Parameters

<i>instance</i>	ADC instance ID.
<i>chnGroup</i>	Selection of configuration group.

Returns

Unformatted conversion value.

3.2.2.15 `int32_t ADC_DRV_ConvRAWData (uint16_t convValue, bool diffEnable, adc_resolution_mode_t mode)`

This function formats the initial value fetched from the ADC module. Initial value fetched from the ADC module can't be read as a number, especially for the signed value generated by the differential conversion. This function can format the initial value to be a readable one.

ADC Peripheral Driver

Parameters

<i>convValue</i>	Initial value directly from the register.
<i>diffEnable</i>	Identifier for the differential mode.
<i>mode</i>	Formatted data resolution mode.

Returns

Formatted conversion value.

3.2.2.16 `adc_status_t ADC_DRV_InstallCallback (uint32_t instance, uint32_t chnGroup, adc_callback_t userCallback)`

This function installs the user-defined callback in the ADC module. When an ADC interrupt request is served, the callback is executed inside the ISR.

Parameters

<i>instance</i>	ADC instance ID.
<i>chnGroup</i>	Selection of the configuration group.
<i>userCallback</i>	User-defined callback function.

Returns

Execution status.

3.2.2.17 `void ADC_DRV_IRQHandler (uint32_t instance)`

This function is the driver-defined ISR in the ADC module. It includes the interrupt mode processes defined by the driver. Currently, it is called inside the system-defined ISR.

Parameters

<i>instance</i>	ADC instance ID.
-----------------	------------------

Chapter 4

Controller Area Network (FlexCAN)

The Kinetis SDK provides both HAL and Peripheral drivers for the FlexCAN Controller Area Network (FlexCAN) block of Kinetis devices.

Modules

- [FlexCAN Driver](#)

This part describes the programming interface of the FlexCAN Peripheral driver.

- [FlexCAN HAL driver](#)

This part describes the programming interface of the FlexCAN HAL driver.

FlexCAN HAL driver

4.1 FlexCAN HAL driver

This chapter describes the programming interface of the FlexCAN HAL driver.

Data Structures

- struct `flexcan_id_table_t`
FlexCAN RX FIFO ID filter table structure. [More...](#)
- struct `flexcan_berr_counter_t`
FlexCAN bus error counters. [More...](#)
- struct `flexcan_mb_code_status_t`
FlexCAN MB code and status for transmit and receive. [More...](#)
- struct `flexcan_mb_t`
FlexCAN message buffer structure. [More...](#)
- struct `flexcan_user_config_t`
FlexCAN configuration. [More...](#)
- struct `flexcan_time_segment_t`
FlexCAN timing related structures. [More...](#)

Enumerations

- enum `_flexcan_constants` { `kFlexCanMessageSize` = 8 }
FlexCAN constants.
- enum `_flexcan_err_status` {
 `kFlexCan_RxWrn` = 0x0080,
 `kFlexCan_TxWrn` = 0x0100,
 `kFlexCan_StfErr` = 0x0200,
 `kFlexCan_FrmErr` = 0x0400,
 `kFlexCan_CrcErr` = 0x0800,
 `kFlexCan_AckErr` = 0x1000,
 `kFlexCan_Bit0Err` = 0x2000,
 `kFlexCan_Bit1Err` = 0x4000 }
- The Status enum is used to report current status of the FlexCAN interface.*
- enum `flexcan_status_t`
FlexCAN status return codes.
- enum `flexcan_operation_modes_t` {
 `kFlexCanNormalMode`,
 `kFlexCanListenOnlyMode`,
 `kFlexCanLoopBackMode`,
 `kFlexCanFreezeMode`,
 `kFlexCanDisableMode` }
- FlexCAN operation modes.*
- enum `flexcan_mb_code_rx_t` {

```
kFlexCanRX_Inactive = 0x0,
kFlexCanRX_Full = 0x2,
kFlexCanRX_Empty = 0x4,
kFlexCanRX_Overrun = 0x6,
kFlexCanRX_Busy = 0x8,
kFlexCanRX_Ranswer = 0xA,
kFlexCanRX_NotUsed = 0xF }
```

FlexCAN message buffer CODE for Rx buffers.

- enum `flexcan_mb_code_tx_t` {
 kFlexCanTX_Inactive = 0x08,
 kFlexCanTX_Abort = 0x09,
 kFlexCanTX_Data = 0x0C,
 kFlexCanTX_Remote = 0x1C,
 kFlexCanTX_Tanswer = 0x0E,
 kFlexCanTX_NotUsed = 0xF }

FlexCAN message buffer CODE FOR Tx buffers.

- enum `flexcan_mb_transmission_type_t` {
 kFlexCanMBStatusType_TX,
 kFlexCanMBStatusType_TXRemote,
 kFlexCanMBStatusType_RX,
 kFlexCanMBStatusType_RXRemote,
 kFlexCanMBStatusType_RXTXRemote }

FlexCAN message buffer transmission types.

- enum `flexcan_rx_fifo_id_element_format_t` {
 kFlexCanRxFifoIdElementFormat_A,
 kFlexCanRxFifoIdElementFormat_B,
 kFlexCanRxFifoIdElementFormat_C,
 kFlexCanRxFifoIdElementFormat_D }
- enum `flexcan_rx_fifo_id_filter_num_t` {
 kFlexCanRxFifoIDFilters_8 = 0x0,
 kFlexCanRxFifoIDFilters_16 = 0x1,
 kFlexCanRxFifoIDFilters_24 = 0x2,
 kFlexCanRxFifoIDFilters_32 = 0x3,
 kFlexCanRxFifoIDFilters_40 = 0x4,
 kFlexCanRxFifoIDFilters_48 = 0x5,
 kFlexCanRxFifoIDFilters_56 = 0x6,
 kFlexCanRxFifoIDFilters_64 = 0x7,
 kFlexCanRxFifoIDFilters_72 = 0x8,
 kFlexCanRxFifoIDFilters_80 = 0x9,
 kFlexCanRxFifoIDFilters_88 = 0xA,
 kFlexCanRxFifoIDFilters_96 = 0xB,
 kFlexCanRxFifoIDFilters_104 = 0xC,
 kFlexCanRxFifoIDFilters_112 = 0xD,
 kFlexCanRxFifoIDFilters_120 = 0xE,
 kFlexCanRxFifoIDFilters_128 = 0xF }

FlexCAN Rx FIFO filters number.

FlexCAN HAL driver

- enum `flexcan_rx_mask_type_t` {
 `kFlexCanRxMask_Global`,
 `kFlexCanRxMask_Individual` }
 FlexCAN RX mask type.
- enum `flexcan_mb_id_type_t` {
 `kFlexCanMbId_Std`,
 `kFlexCanMbId_Ext` }
 FlexCAN MB ID type.
- enum `flexcan_clk_source_t` {
 `kFlexCanClkSource_Osc`,
 `kFlexCanClkSource_Ipbus` }
 FlexCAN clock source.
- enum `flexcan_int_type_t` {
 `kFlexCanInt_Buf`,
 `kFlexCanInt_Err`,
 `kFlexCanInt_Boff`,
 `kFlexCanInt_Wakeup`,
 `kFlexCanInt_Txwarning`,
 `kFlexCanInt_Rxwarning` }
 FlexCAN error interrupt types.

Configuration

- `flexcan_status_t FLEXCAN_HAL_Enable` (`uint32_t canBaseAddr`)
 Enables FlexCAN controller.
- `flexcan_status_t FLEXCAN_HAL_Disable` (`uint32_t canBaseAddr`)
 Disables FlexCAN controller.
- static `bool FLEXCAN_HAL_IsEnabled` (`uint32_t canBaseAddr`)
 Checks whether the FlexCAN is enabled or disabled.
- `flexcan_status_t FLEXCAN_HAL_SelectClock` (`uint32_t canBaseAddr, flexcan_clk_source_t clk`)
 Selects the clock source for FlexCAN.
- `flexcan_status_t FLEXCAN_HAL_Init` (`uint32_t canBaseAddr, const flexcan_user_config_t *data`)
 Initializes the FlexCAN controller.
- `void FLEXCAN_HAL_SetTimeSegments` (`uint32_t canBaseAddr, flexcan_time_segment_t *time-seg`)
 Sets the FlexCAN time segments for setting up bit rate.
- `void FLEXCAN_HAL_GetTimeSegments` (`uint32_t canBaseAddr, flexcan_time_segment_t *time-seg`)
 Gets the FlexCAN time segments to calculate the bit rate.
- `void FLEXCAN_HAL_ExitFreezeMode` (`uint32_t canBaseAddr`)
 Un freezes the FlexCAN module.
- `void FLEXCAN_HAL_EnterFreezeMode` (`uint32_t canBaseAddr`)
 Freezes the FlexCAN module.
- `flexcan_status_t FLEXCAN_HAL_EnableOperationMode` (`uint32_t canBaseAddr, flexcan_operation_modes_t mode`)
 Enables operation mode.
- `flexcan_status_t FLEXCAN_HAL_DisableOperationMode` (`uint32_t canBaseAddr, flexcan_operation_modes_t mode`)

Disables operation mode.

Data transfer

- `flexcan_status_t FLEXCAN_HAL_SetMbTx (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx, flexcan_mb_code_status_t *cs, uint32_t msg_id, uint8_t *mb_data)`
Sets the FlexCAN message buffer fields for transmitting.
- `flexcan_status_t FLEXCAN_HAL_SetMbRx (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx, flexcan_mb_code_status_t *cs, uint32_t msg_id)`
Sets the FlexCAN message buffer fields for receiving.
- `flexcan_status_t FLEXCAN_HAL_GetMb (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx, flexcan_mb_t *mb)`
Gets the FlexCAN message buffer fields.
- `flexcan_status_t FLEXCAN_HAL_LockRxMb (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx)`
Locks the FlexCAN Rx message buffer.
- static void `FLEXCAN_HAL_UnlockRxMb (uint32_t canBaseAddr)`
Unlocks the FlexCAN Rx message buffer.
- void `FLEXCAN_HAL_EnableRxFifo (uint32_t canBaseAddr)`
Enables the Rx FIFO.
- void `FLEXCAN_HAL_DisableRxFifo (uint32_t canBaseAddr)`
Disables the Rx FIFO.
- void `FLEXCAN_HAL_SetRxFifoFiltersNumber (uint32_t canBaseAddr, uint32_t number)`
Sets the number of the Rx FIFO filters.
- void `FLEXCAN_HAL_SetMaxMbNumber (uint32_t canBaseAddr, const flexcan_user_config_t *data)`
Sets the maximum number of Message Buffers.
- `flexcan_status_t FLEXCAN_HAL_SetIdFilterTableElements (uint32_t canBaseAddr, const flexcan_user_config_t *data, flexcan_rx_fifo_id_element_format_t id_format, flexcan_id_table_t *id_filter_table)`
Sets the Rx FIFO ID filter table elements.
- `flexcan_status_t FLEXCAN_HAL_SetRxFifo (uint32_t canBaseAddr, const flexcan_user_config_t *data, flexcan_rx_fifo_id_element_format_t id_format, flexcan_id_table_t *id_filter_table)`
Sets the FlexCAN Rx FIFO fields.
- `flexcan_status_t FLEXCAN_HAL_ReadFifo (uint32_t canBaseAddr, flexcan_mb_t *rx_fifo)`
Gets the FlexCAN Rx FIFO data.

Interrupts

- `flexcan_status_t FLEXCAN_HAL_EnableMbInt (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx)`
Enables the FlexCAN Message Buffer interrupt.
- `flexcan_status_t FLEXCAN_HAL_DisableMbInt (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx)`
Disables the FlexCAN Message Buffer interrupt.
- void `FLEXCAN_HAL_EnableErrInt (uint32_t canBaseAddr)`
Enables error interrupt of the FlexCAN module.

FlexCAN HAL driver

- void **FLEXCAN_HAL_DisableErrInt** (uint32_t canBaseAddr)
Disables error interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_EnableBusOffInt** (uint32_t canBaseAddr)
Enables Bus off interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_DisableBusOffInt** (uint32_t canBaseAddr)
Disables Bus off interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_EnableWakeupInt** (uint32_t canBaseAddr)
Enables Wakeup interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_DisableWakeupInt** (uint32_t canBaseAddr)
Disables Wakeup interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_EnableTxWarningInt** (uint32_t canBaseAddr)
Enables TX warning interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_DisableTxWarningInt** (uint32_t canBaseAddr)
Disables TX warning interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_EnableRxWarningInt** (uint32_t canBaseAddr)
Enables RX warning interrupt of the FlexCAN module.
- void **FLEXCAN_HAL_DisableRxWarningInt** (uint32_t canBaseAddr)
Disables RX warning interrupt of the FlexCAN module.

Status

- static uint32_t **FLEXCAN_HAL_GetFreezeAck** (uint32_t canBaseAddr)
Gets the value of FlexCAN freeze ACK.
- uint8_t **FLEXCAN_HAL_GetMbIntFlag** (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx)
Gets the individual FlexCAN MB interrupt flag.
- static uint32_t **FLEXCAN_HAL_GetAllMbIntFlags** (uint32_t canBaseAddr)
Gets all FlexCAN MB interrupt flags.
- static void **FLEXCAN_HAL_ClearMbIntFlag** (uint32_t canBaseAddr, uint32_t reg_val)
Clears the interrupt flag of the message buffers.
- void **FLEXCAN_HAL_GetErrCounter** (uint32_t canBaseAddr, flexcan_berr_counter_t *err_cnt)
Gets the transmit error counter and receives the error counter.
- static uint32_t **FLEXCAN_HAL_GetErrStatus** (uint32_t canBaseAddr)
Gets error and status.
- void **FLEXCAN_HAL_ClearErrIntStatus** (uint32_t canBaseAddr)
Clears all other interrupts in ERRSTAT register (Error, Busoff, Wakeup).

Mask

- void **FLEXCAN_HAL_SetMaskType** (uint32_t canBaseAddr, flexcan_rx_mask_type_t type)
Sets the Rx masking type.
- void **FLEXCAN_HAL_SetRx_fifoGlobalStdMask** (uint32_t canBaseAddr, uint32_t std_mask)
Sets the FlexCAN RX FIFO global standard mask.
- void **FLEXCAN_HAL_SetRx_fifoGlobalExtMask** (uint32_t canBaseAddr, uint32_t ext_mask)
Sets the FlexCAN Rx FIFO global extended mask.
- flexcan_status_t **FLEXCAN_HAL_SetRxIndividualStdMask** (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx, uint32_t std_mask)
Sets the FlexCAN Rx individual standard mask for ID filtering in the Rx MBs and the Rx FIFO.

- `flexcan_status_t FLEXCAN_HAL_SetRxIndividualExtMask (uint32_t canBaseAddr, const flexcan_user_config_t *data, uint32_t mb_idx, uint32_t ext_mask)`
Sets the FlexCAN Rx individual extended mask for ID filtering in the Rx MBs and the Rx FIFO.
- `void FLEXCAN_HAL_SetRxMbGlobalStdMask (uint32_t canBaseAddr, uint32_t std_mask)`
Sets the FlexCAN Rx MB global standard mask.
- `void FLEXCAN_HAL_SetRxMbBuf14StdMask (uint32_t canBaseAddr, uint32_t std_mask)`
Sets the FlexCAN RX MB BUF14 standard mask.
- `void FLEXCAN_HAL_SetRxMbBuf15StdMask (uint32_t canBaseAddr, uint32_t std_mask)`
Sets the FlexCAN RX MB BUF15 standard mask.
- `void FLEXCAN_HAL_SetRxMbGlobalExtMask (uint32_t canBaseAddr, uint32_t ext_mask)`
Sets the FlexCAN RX MB global extended mask.
- `void FLEXCAN_HAL_SetRxMbBuf14ExtMask (uint32_t canBaseAddr, uint32_t ext_mask)`
Sets the FlexCAN RX MB BUF14 extended mask.
- `void FLEXCAN_HAL_SetRxMbBuf15ExtMask (uint32_t canBaseAddr, uint32_t ext_mask)`
Sets the FlexCAN RX MB BUF15 extended mask.
- `static uint32_t FLEXCAN_HAL_GetIdAcceptanceFilterRxFifo (uint32_t canBaseAddr)`
Gets the FlexCAN ID acceptance filter hit indicator on Rx FIFO.

4.1.1 Data Structure Documentation

4.1.1.1 struct flexcan_id_table_t

Data Fields

- `bool is_remote_mb`
Remote frame.
- `bool is_extended_mb`
Extended frame.
- `uint32_t * id_filter`
Rx FIFO ID filter elements.

4.1.1.2 struct flexcan_berr_counter_t

Data Fields

- `uint16_t txerr`
Transmit error counter.
- `uint16_t rxerr`
Receive error counter.

4.1.1.3 struct flexcan_mb_code_status_t

Data Fields

- `uint32_t code`
MB code for TX or RX buffers.
- `flexcan_mb_id_type_t msg_id_type`

FlexCAN HAL driver

- `uint32_t data_length`
Type of message ID (standard or extended)
Length of Data in Bytes.

4.1.1.3.0.5 Field Documentation

4.1.1.3.0.5.1 `uint32_t flexcan_mb_code_status_t::code`

Defined by `flexcan_mb_code_rx_t` and `flexcan_mb_code_tx_t`

4.1.1.4 `struct flexcan_mb_t`

Data Fields

- `uint32_t cs`
Code and Status.
- `uint32_t msg_id`
Message Buffer ID.
- `uint8_t data [kFlexCanMessageSize]`
Bytes of the FlexCAN message.

4.1.1.5 `struct flexcan_user_config_t`

Data Fields

- `uint32_t max_num_mb`
The maximum number of Message Buffers.
- `flexcan_rx_fifo_id_filter_num_t num_id_filters`
The number of Rx FIFO ID filters needed.
- `bool is_rx_fifo_needed`
1 if needed; 0 if not

4.1.1.6 `struct flexcan_time_segment_t`

Data Fields

- `uint32_t propseg`
Propagation segment.
- `uint32_t pseg1`
Phase segment 1.
- `uint32_t pseg2`
Phase segment 2.
- `uint32_t pre_divider`
Clock pre divider.
- `uint32_t rjw`
Resync jump width.

4.1.2 Enumeration Type Documentation

4.1.2.1 enum _flexcan_constants

Enumerator

kFlexCanMessageSize FlexCAN message buffer data size in bytes.

4.1.2.2 enum _flexcan_err_status

Enumerator

kFlexCan_RxWrn Reached warning level for RX errors.

kFlexCan_TxWrn Reached warning level for TX errors.

kFlexCan_StfErr Stuffing Error.

kFlexCan_FrmErr Form Error.

kFlexCan_CrcErr Cyclic Redundancy Check Error.

kFlexCan_AckErr Received no ACK on transmission.

kFlexCan_Bit0Err Unable to send dominant bit.

kFlexCan_Bit1Err Unable to send recessive bit.

4.1.2.3 enum flexcan_operation_modes_t

Enumerator

kFlexCanNormalMode Normal mode or user mode.

kFlexCanListenOnlyMode Listen-only mode.

kFlexCanLoopBackMode Loop-back mode.

kFlexCanFreezeMode Freeze mode.

kFlexCanDisableMode Module disable mode.

4.1.2.4 enum flexcan_mb_code_rx_t

Enumerator

kFlexCanRX_Inactive MB is not active.

kFlexCanRX_Full MB is full.

kFlexCanRX_Empty MB is active and empty.

kFlexCanRX_Overrun MB is overwritten into a full buffer.

kFlexCanRX_Busy FlexCAN is updating the contents of the MB.

kFlexCanRX_Ranswer The CPU must not access the MB. A frame was configured to recognize a Remote Request Frame

kFlexCanRX_NotUsed and transmit a Response Frame in return. Not used

FlexCAN HAL driver

4.1.2.5 enum flexcan_mb_code_tx_t

Enumerator

kFlexCanTX_Inactive MB is not active.

kFlexCanTX_Abort MB is aborted.

kFlexCanTX_Data MB is a TX Data Frame(MB RTR must be 0).

kFlexCanTX_Remote MB is a TX Remote Request Frame (MB RTR must be 1).

kFlexCanTX_Tanswer MB is a TX Response Request Frame from.

kFlexCanTX_NotUsed an incoming Remote Request Frame. Not used

4.1.2.6 enum flexcan_mb_transmission_type_t

Enumerator

kFlexCanMBStatusType_TX Transmit MB.

kFlexCanMBStatusType_TXRemote Transmit remote request MB.

kFlexCanMBStatusType_RX Receive MB.

kFlexCanMBStatusType_RXRemote Receive remote request MB.

kFlexCanMBStatusType_RXTXRemote FlexCAN remote frame receives remote request and.

4.1.2.7 enum flexcan_rx_fifo_id_element_format_t

Enumerator

kFlexCanRxFifoIdElementFormat_A One full ID (standard and extended) per ID Filter Table.

kFlexCanRxFifoIdElementFormat_B element. Two full standard IDs or two partial 14-bit (standard and

kFlexCanRxFifoIdElementFormat_C extended) IDs per ID Filter Table element. Four partial 8-bit Standard IDs per ID Filter Table

kFlexCanRxFifoIdElementFormat_D element. All frames rejected.

4.1.2.8 enum flexcan_rx_fifo_id_filter_num_t

Enumerator

kFlexCanRxFifoIDFilters_8 8 Rx FIFO Filters

kFlexCanRxFifoIDFilters_16 16 Rx FIFO Filters

kFlexCanRxFifoIDFilters_24 24 Rx FIFO Filters

kFlexCanRxFifoIDFilters_32 32 Rx FIFO Filters

kFlexCanRxFifoIDFilters_40 40 Rx FIFO Filters

kFlexCanRxFifoIDFilters_48 48 Rx FIFO Filters

kFlexCanRxFifoIDFilters_56 56 Rx FIFO Filters

kFlexCanRxFifoIDFilters_64 64 Rx FIFO Filters

kFlexCanRxFifoIDFilters_72 72 Rx FIFO Filters
kFlexCanRxFifoIDFilters_80 80 Rx FIFO Filters
kFlexCanRxFifoIDFilters_88 88 Rx FIFO Filters
kFlexCanRxFifoIDFilters_96 96 Rx FIFO Filters
kFlexCanRxFifoIDFilters_104 104 Rx FIFO Filters
kFlexCanRxFifoIDFilters_112 112 Rx FIFO Filters
kFlexCanRxFifoIDFilters_120 120 Rx FIFO Filters
kFlexCanRxFifoIDFilters_128 128 Rx FIFO Filters

4.1.2.9 enum flexcan_rx_mask_type_t

Enumerator

kFlexCanRxMask_Global Rx global mask.
kFlexCanRxMask_Individual Rx individual mask.

4.1.2.10 enum flexcan_mb_id_type_t

Enumerator

kFlexCanMbId_Std Standard ID.
kFlexCanMbId_Ext Extended ID.

4.1.2.11 enum flexcan_clk_source_t

Enumerator

kFlexCanClkSource_Osc Oscillator clock.
kFlexCanClkSource_Ipbus Peripheral clock.

4.1.2.12 enum flexcan_int_type_t

Enumerator

kFlexCanInt_Buf OR'd message buffers interrupt.
kFlexCanInt_Err Error interrupt.
kFlexCanInt_Boff Bus off interrupt.
kFlexCanInt_Wakeup Wakeup interrupt.
kFlexCanInt_Txwarning TX warning interrupt.
kFlexCanInt_Rxwarning RX warning interrupt.

4.1.3 Function Documentation

4.1.3.1 `flexcan_status_t FLEXCAN_HAL_Enable(uint32_t canBaseAddr)`

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

0 if successful; non-zero failed

4.1.3.2 **flexcan_status_t FLEXCAN_HAL_Disable (uint32_t canBaseAddr)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

0 if successful; non-zero failed

4.1.3.3 **static bool FLEXCAN_HAL_IsEnabled (uint32_t canBaseAddr) [inline], [static]**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

State of FlexCAN enable(0)/disable(1)

4.1.3.4 **flexcan_status_t FLEXCAN_HAL_SelectClock (uint32_t canBaseAddr, flexcan_clk_source_t clk)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

FlexCAN HAL driver

<i>clk</i>	The FlexCAN clock source
------------	--------------------------

Returns

0 if successful; non-zero failed

4.1.3.5 **flexcan_status_t FLEXCAN_HAL_Init (uint32_t canBaseAddr, const flexcan_user_config_t * data)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data.

Returns

0 if successful; non-zero failed

4.1.3.6 **void FLEXCAN_HAL_SetTimeSegments (uint32_t canBaseAddr, flexcan_time_segment_t * time_seg)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>time_seg</i>	FlexCAN time segments, which need to be set for the bit rate.

Returns

0 if successful; non-zero failed

4.1.3.7 **void FLEXCAN_HAL_GetTimeSegments (uint32_t canBaseAddr, flexcan_time_segment_t * time_seg)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>time_seg</i>	FlexCAN time segments read for bit rate

Returns

0 if successful; non-zero failed.

4.1.3.8 void FLEXCAN_HAL_ExitFreezeMode (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

0 if successful; non-zero failed.

4.1.3.9 void FLEXCAN_HAL_EnterFreezeMode (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.10 flexcan_status_t FLEXCAN_HAL_EnableOperationMode (uint32_t *canBaseAddr*, flexcan_operation_modes_t *mode*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>mode</i>	An operation mode to be enabled

Returns

0 if successful; non-zero failed.

4.1.3.11 flexcan_status_t FLEXCAN_HAL_DisableOperationMode (uint32_t *canBaseAddr*, flexcan_operation_modes_t *mode*)

FlexCAN HAL driver

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>mode</i>	An operation mode to be disabled

Returns

0 if successful; non-zero failed.

4.1.3.12 flexcan_status_t FLEXCAN_HAL_SetMbTx (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, flexcan_mb_code_status_t * *cs*, uint32_t *msg_id*, uint8_t * *mb_data*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>cs</i>	CODE/status values (TX)
<i>msg_id</i>	ID of the message to transmit
<i>mb_data</i>	Bytes of the FlexCAN message

Returns

0 if successful; non-zero failed

4.1.3.13 flexcan_status_t FLEXCAN_HAL_SetMbRx (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, flexcan_mb_code_status_t * *cs*, uint32_t *msg_id*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>cs</i>	CODE/status values (RX)
<i>msg_id</i>	ID of the message to receive

Returns

0 if successful; non-zero failed

4.1.3.14 **flexcan_status_t FLEXCAN_HAL_GetMb (uint32_t canBaseAddr, const flexcan_user_config_t * data, uint32_t mb_idx, flexcan_mb_t * mb)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>mb</i>	The fields of the message buffer

Returns

0 if successful; non-zero failed

4.1.3.15 **flexcan_status_t FLEXCAN_HAL_LockRxMb (uint32_t canBaseAddr, const flexcan_user_config_t * data, uint32_t mb_idx)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer

Returns

0 if successful; non-zero failed

4.1.3.16 **static void FLEXCAN_HAL_UnlockRxMb (uint32_t canBaseAddr) [inline], [static]**

FlexCAN HAL driver

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

0 if successful; non-zero failed

4.1.3.17 void FLEXCAN_HAL_EnableRxFifo (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.18 void FLEXCAN_HAL_DisableRxFifo (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.19 void FLEXCAN_HAL_SetRx_fifoFiltersNumber (uint32_t *canBaseAddr*, uint32_t *number*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>number</i>	The number of Rx FIFO filters

4.1.3.20 void FLEXCAN_HAL_SetMaxMbNumber (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data

4.1.3.21 flexcan_status_t FLEXCAN_HAL_SetIdFilterTableElements (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*, flexcan_rx_fifo_id_element_format_t *id_format*, flexcan_id_table_t * *id_filter_table*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>id_format</i>	The format of the Rx FIFO ID Filter Table Elements
<i>id_filter_table</i>	The ID filter table elements which contain if RTR bit, IDE bit and RX message ID need to be set.

Returns

0 if successful; non-zero failed.

4.1.3.22 flexcan_status_t FLEXCAN_HAL_SetRxFifo (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*, flexcan_rx_fifo_id_element_format_t *id_format*, flexcan_id_table_t * *id_filter_table*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>id_format</i>	The format of the Rx FIFO ID Filter Table Elements
<i>id_filter_table</i>	The ID filter table elements which contain RTR bit, IDE bit, and RX message ID.

Returns

0 if successful; non-zero failed.

4.1.3.23 flexcan_status_t FLEXCAN_HAL_ReadFifo (uint32_t *canBaseAddr*, flexcan_mb_t * *rx_fifo*)

FlexCAN HAL driver

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>rx_fifo</i>	The FlexCAN receive FIFO data

Returns

0 if successful; non-zero failed.

4.1.3.24 **flexcan_status_t FLEXCAN_HAL_EnableMbInt (uint32_t canBaseAddr, const flexcan_user_config_t * data, uint32_t mb_idx)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer

Returns

0 if successful; non-zero failed

4.1.3.25 **flexcan_status_t FLEXCAN_HAL_DisableMbInt (uint32_t canBaseAddr, const flexcan_user_config_t * data, uint32_t mb_idx)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer

Returns

0 if successful; non-zero failed

4.1.3.26 **void FLEXCAN_HAL_EnableErrInt (uint32_t canBaseAddr)**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.27 void FLEXCAN_HAL_DisableErrInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.28 void FLEXCAN_HAL_EnableBusOffInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.29 void FLEXCAN_HAL_DisableBusOffInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.30 void FLEXCAN_HAL_EnableWakeuInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.31 void FLEXCAN_HAL_DisableWakeuInt (uint32_t *canBaseAddr*)

Parameters

FlexCAN HAL driver

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.32 void FLEXCAN_HAL_EnableTxWarningInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.33 void FLEXCAN_HAL_DisableTxWarningInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.34 void FLEXCAN_HAL_EnableRxWarningInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.35 void FLEXCAN_HAL_DisableRxWarningInt (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.36 static uint32_t FLEXCAN_HAL_GetFreezeAck (uint32_t *canBaseAddr*) [inline], [static]

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

freeze ACK state (1-freeze mode, 0-not in freeze mode).

4.1.3.37 `uint8_t FLEXCAN_HAL_GetMbIntFlag (uint32_t canBaseAddr, const flexcan_user_config_t * data, uint32_t mb_idx)`

FlexCAN HAL driver

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer

Returns

the individual MB interrupt flag (0 and 1 are the flag value)

**4.1.3.38 static uint32_t FLEXCAN_HAL_GetAllMbIntFlags (uint32_t *canBaseAddr*)
[inline], [static]**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

all MB interrupt flags

4.1.3.39 static void FLEXCAN_HAL_ClearMbIntFlag (uint32_t *canBaseAddr*, uint32_t *reg_val*) [inline], [static]

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>reg_val</i>	The value to be written to the interrupt flag1 register.

4.1.3.40 void FLEXCAN_HAL_GetErrCounter (uint32_t *canBaseAddr*, flexcan_berr_counter_t * *err_cnt*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>err_cnt</i>	Transmit error counter and receive error counter

**4.1.3.41 static uint32_t FLEXCAN_HAL_GetErrStatus (uint32_t *canBaseAddr*)
[inline], [static]**

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

The current error and status

4.1.3.42 void FLEXCAN_HAL_ClearErrIntStatus (uint32_t *canBaseAddr*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

4.1.3.43 void FLEXCAN_HAL_SetMaskType (uint32_t *canBaseAddr*, flexcan_rx_mask_type_t *type*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>type</i>	The FlexCAN Rx mask type

4.1.3.44 void FLEXCAN_HAL_SetRx_fifoGlobalStdMask (uint32_t *canBaseAddr*, uint32_t *std_mask*)

Parameters

FlexCAN HAL driver

<i>canBaseAddr</i>	The FlexCAN base address
<i>std_mask</i>	Standard mask

4.1.3.45 void FLEXCAN_HAL_SetRxFifoGlobalExtMask (uint32_t *canBaseAddr*, uint32_t *ext_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>ext_mask</i>	Extended mask

4.1.3.46 flexcan_status_t FLEXCAN_HAL_SetRxIndividualStdMask (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, uint32_t *std_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>std_mask</i>	Individual standard mask

Returns

0 if successful; non-zero failed

4.1.3.47 flexcan_status_t FLEXCAN_HAL_SetRxIndividualExtMask (uint32_t *canBaseAddr*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, uint32_t *ext_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>ext_mask</i>	Individual extended mask

Returns

0 if successful; non-zero failed

4.1.3.48 void FLEXCAN_HAL_SetRxMbGlobalStdMask (uint32_t *canBaseAddr*, uint32_t *std_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>std_mask</i>	Standard mask

4.1.3.49 void FLEXCAN_HAL_SetRxMbBuf14StdMask (uint32_t *canBaseAddr*, uint32_t *std_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>std_mask</i>	Standard mask

4.1.3.50 void FLEXCAN_HAL_SetRxMbBuf15StdMask (uint32_t *canBaseAddr*, uint32_t *std_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>std_mask</i>	Standard mask

Returns

0 if successful; non-zero failed

4.1.3.51 void FLEXCAN_HAL_SetRxMbGlobalExtMask (uint32_t *canBaseAddr*, uint32_t *ext_mask*)

FlexCAN HAL driver

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>ext_mask</i>	Extended mask

4.1.3.52 void FLEXCAN_HAL_SetRxMbBuf14ExtMask (uint32_t *canBaseAddr*, uint32_t *ext_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>ext_mask</i>	Extended mask

4.1.3.53 void FLEXCAN_HAL_SetRxMbBuf15ExtMask (uint32_t *canBaseAddr*, uint32_t *ext_mask*)

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
<i>ext_mask</i>	Extended mask

4.1.3.54 static uint32_t FLEXCAN_HAL_GetIdAcceptanceFilterRxFifo (uint32_t *canBaseAddr*) [inline], [static]

Parameters

<i>canBaseAddr</i>	The FlexCAN base address
--------------------	--------------------------

Returns

RX FIFO information

4.2 FlexCAN Driver

This chapter describes the programming interface of the FlexCAN Peripheral driver.

Data Structures

- struct `flexcan_bitrate_table_t`
FlexCAN bit rate and the related timing segments structure. [More...](#)
- struct `flexcan_data_info_t`
FlexCAN data info from user. [More...](#)

Enumerations

- enum `flexcan_bitrate_t` {

`kFlexCanBitrate_125k` = 125000,
`kFlexCanBitrate_250k` = 250000,
`kFlexCanBitrate_500k` = 500000,
`kFlexCanBitrate_750k` = 750000,
`kFlexCanBitrate_1M` = 1000000 }

FlexCAN bitrates supported.

Functions

- void `FLEXCAN_DRV_IRQHandler` (uint8_t instance)
Interrupt handler for a FlexCAN instance.

Bit rate

- `flexcan_status_t FLEXCAN_DRV_SetBitrate` (uint8_t instance, `flexcan_bitrate_t` bitrate)
Sets the FlexCAN bit rate.
- `flexcan_status_t FLEXCAN_DRV_GetBitrate` (uint8_t instance, `flexcan_bitrate_t` *bitrate)
Gets the FlexCAN bit rate.

Global mask

- void `FLEXCAN_DRV_SetMaskType` (uint8_t instance, `flexcan_rx_mask_type_t` type)
Sets the RX masking type.
- `flexcan_status_t FLEXCAN_DRV_SetRx_fifoGlobalMask` (uint8_t instance, `flexcan_mb_id_type_t` id_type, uint32_t mask)
Sets the FlexCAN RX FIFO global standard or extended mask.
- `flexcan_status_t FLEXCAN_DRV_SetRxMbGlobalMask` (uint8_t instance, `flexcan_mb_id_type_t` id_type, uint32_t mask)
Sets the FlexCAN RX MB global standard or extended mask.

FlexCAN Driver

- `flexcan_status_t FLEXCAN_DRV_SetRxIndividualMask` (uint8_t instance, const `flexcan_user_config_t` *data, `flexcan_mb_id_type_t` id_type, uint32_t mb_idx, uint32_t mask)
Sets the FlexCAN RX individual standard or extended mask.

Initialization and Shutdown

- `flexcan_status_t FLEXCAN_DRV_Init` (uint8_t instance, const `flexcan_user_config_t` *data, bool enable_err_interrupts)
Initializes the FlexCAN peripheral.
- `uint32_t FLEXCAN_DRV_Deinit` (uint8_t instance)
Shuts down a FlexCAN instance.

Send configuration

- `flexcan_status_t FLEXCAN_DRV_ConfigTxMb` (uint8_t instance, const `flexcan_user_config_t` *data, uint32_t mb_idx, `flexcan_data_info_t` *tx_info, uint32_t msg_id)
FlexCAN transmit message buffer field configuration.
- `flexcan_status_t FLEXCAN_DRV_Send` (uint8_t instance, const `flexcan_user_config_t` *data, uint32_t mb_idx, `flexcan_data_info_t` *tx_info, uint32_t msg_id, uint8_t *mb_data)
Sends FlexCAN messages.

Receive configuration

- `flexcan_status_t FLEXCAN_DRV_ConfigRxMb` (uint8_t instance, const `flexcan_user_config_t` *data, uint32_t mb_idx, `flexcan_data_info_t` *rx_info, uint32_t msg_id)
FlexCAN receive message buffer field configuration.
- `flexcan_status_t FLEXCAN_DRV_ConfigRxFifo` (uint8_t instance, const `flexcan_user_config_t` *data, `flexcan_rx_fifo_id_element_format_t` id_format, `flexcan_id_table_t` *id_filter_table)
FlexCAN RX FIFO field configuration.
- `flexcan_status_t FLEXCAN_DRV_RxMessageBuffer` (uint8_t instance, const `flexcan_user_config_t` *config, uint32_t mb_idx, `flexcan_mb_t` *data)
FlexCAN is waiting to receive data from the Message buffer.
- `flexcan_status_t FLEXCAN_DRV_RxFifo` (uint8_t instance, const `flexcan_user_config_t` *config, `flexcan_mb_t` *data)
FlexCAN is waiting to receive data from the Message FIFO.

4.2.0.55 FlexCAN Driver

Overview

The FlexCAN (flexible controller area network) module is a communication controller implementing the CAN protocol according to the CAN 2.0B protocol specification. The FlexCAN module supports both standard and extended message frames. The Message Buffers are stored in an embedded RAM dedicated

to the FlexCAN module. The CAN Protocol Engine (PE) sub-module manages the serial communication on the CAN bus by requesting RAM access for receiving and transmitting message frames, validating received messages, and performing error handling.

Initialization

To initialize the FlexCAN driver, call the [FLEXCAN_DRV_Init\(\)](#) function and pass the instance number of the FlexCAN you want to use. For example, to use FlexCAN0, pass a value of 0 to the `flexcan_init` function. In addition, you should also pass a user configuration structure [flexcan_user_config_t](#), as shown here:

```
// FlexCAN configuration structure for user
typedef struct FLEXCANUserConfig {
    uint32_t max_num_mb;
    flexcan_rx_fifo_id_filter_num_t num_id_filters;
    bool is_rx_fifo_needed;
} flexcan_user_config_t;
```

Typically, the user configures the [flexcan_user_config_t](#) instantiation as 16 message buffers needed, 8 RX FIFO ID filters and set to true when RX FIFO is needed. This is a code example to set up a FlexCAN configuration instantiation:

```
flexcan_config_t flexcan1_data;

flexcan1_data.max_num_mb = 16;
flexcan1_data.num_id_filters = kFlexCanRx_fifoIDFilters_8;
flexcan1_data.is_rx_fifo_needed = true;
```

Module timing

FlexCAN bit rate is derived from the serial clock, which is generated by dividing the PE clock by the programmed PRESDIV value. Each serial clock period is also called a time quantum. The FlexCAN bit-rate is defined as the Sclock divided by the number of time quanta, where time quanta are further broken down segments within the bit time (time to transmit and sample a bit). The following list shows the CAN bit-rates that are supported in the FlexCAN driver.

- 1 Mbytes/s
- 750 Kbytes/s
- 500 Kbytes/s
- 250 Kbytes/s
- 125 Kbytes/s

The FlexCAN module supports several different ways to set up the bit timing parameters that are required by the CAN protocol. The Control Register has various fields used to control bit timing parameters: PRESDIV, PROPSEG, PSEG1, PSEG2, and RJW.

To calculate the CAN bit timing parameters, use the method outlined in the Application Note AN1798, chapter 4.1. A maximum time for PROP_SEG is used, the remaining TQ is split equally between PSEG1 and PSEG2, provided PSEG2 ≥ 2 . RJW is set to the minimum of 4 or to the PSEG1.

Transfers

To transmit a FlexCAN frame, the CPU must prepare a message buffer for transmission by calling the [FLEXCAN_DRV_Send\(\)](#) function. To receive the FlexCAN frames into a message buffer, the CPU must also prepare it for reception by first setting up the receive mask. Functions for this are [FLEXCAN_DRV_SetMaskType\(\)](#), [FLEXCAN_DRV_SetRxFifoGlobalMask\(\)](#), [FLEXCAN_DRV_SetRxMbGlobalMask\(\)](#) and [FLEXCAN_DRV_SetRxIndividualMask\(\)](#). The user can then configure the RX buffers by calling either [FLEXCAN_DRV_ConfigRxMb\(\)](#) or [FLEXCAN_DRV_ConfigRxFifo\(\)](#) depending on the mode of reception. Finally the user will call [FLEXCAN_DRV_RxMessageBuffer\(\)](#) or [FLEXCAN_DRV_RxFifo](#) functions depending on the mode of reception. The FlexCAN uses only the interrupt-driven process to transfer data.

4.2.1 Data Structure Documentation

4.2.1.1 struct flexcan_bitrate_table_t

Data Fields

- [flexcan_bitrate_t bit_rate](#)
bit rate
- [uint32_t propseg](#)
Propagation segment.
- [uint32_t pseg1](#)
Phase segment 1.
- [uint32_t pseg2](#)
Phase segment 2.
- [uint32_t pre_divider](#)
Clock pre divider.
- [uint32_t rjw](#)
Re-sync jump width.

4.2.1.2 struct flexcan_data_info_t

Data Fields

- [flexcan_mb_id_type_t msg_id_type](#)
Type of message ID (standard or extended)
- [uint32_t data_length](#)
Length of Data in Bytes.

4.2.2 Enumeration Type Documentation

4.2.2.1 enum flexcan_bitrate_t

Enumerator

<i>kFlexCanBitrate_125k</i>	125 kHz
<i>kFlexCanBitrate_250k</i>	250 kHz
<i>kFlexCanBitrate_500k</i>	500 kHz
<i>kFlexCanBitrate_750k</i>	750 kHz
<i>kFlexCanBitrate_1M</i>	1 MHz

4.2.3 Function Documentation

4.2.3.1 flexcan_status_t FLEXCAN_DRV_SetBitrate (uint8_t *instance*, flexcan_bitrate_t *bitrate*)

Parameters

<i>instance</i>	A FlexCAN instance number
<i>bitrate</i>	Selects a FlexCAN bit rate in the bit_rate_table.

Returns

0 if successful; non-zero failed

4.2.3.2 flexcan_status_t FLEXCAN_DRV_GetBitrate (uint8_t *instance*, flexcan_bitrate_t * *bitrate*)

Parameters

<i>instance</i>	A FlexCAN instance number
<i>bitrate</i>	A pointer to a variable for returning the FlexCAN bit rate in the bit_rate_table.

Returns

0 if successful; non-zero failed

4.2.3.3 void FLEXCAN_DRV_SetMaskType (uint8_t *instance*, flexcan_rx_mask_type_t *type*)

FlexCAN Driver

Parameters

<i>instance</i>	A FlexCAN instance number
<i>type</i>	The FlexCAN RX mask type

4.2.3.4 **flexcan_status_t FLEXCAN_DRV_SetRxFifoGlobalMask (uint8_t *instance*, flexcan_mb_id_type_t *id_type*, uint32_t *mask*)**

Parameters

<i>instance</i>	A FlexCAN instance number
<i>id_type</i>	Standard ID or extended ID
<i>mask</i>	Mask value

Returns

0 if successful; non-zero failed

4.2.3.5 **flexcan_status_t FLEXCAN_DRV_SetRxMbGlobalMask (uint8_t *instance*, flexcan_mb_id_type_t *id_type*, uint32_t *mask*)**

Parameters

<i>instance</i>	A FlexCAN instance number
<i>id_type</i>	Standard ID or extended ID
<i>mask</i>	Mask value

Returns

0 if successful; non-zero failed

4.2.3.6 **flexcan_status_t FLEXCAN_DRV_SetRxIndividualMask (uint8_t *instance*, const flexcan_user_config_t * *data*, flexcan_mb_id_type_t *id_type*, uint32_t *mb_idx*, uint32_t *mask*)**

Parameters

<i>instance</i>	A FlexCAN instance number
<i>data</i>	The FlexCAN platform data
<i>id_type</i>	A standard ID or an extended ID
<i>mb_idx</i>	Index of the message buffer
<i>mask</i>	Mask value

Returns

0 if successful; non-zero failed.

4.2.3.7 **flexcan_status_t FLEXCAN_DRV_Init (uint8_t *instance*, const flexcan_user_config_t * *data*, bool *enable_err_interrupts*)**

This function initializes

Parameters

<i>instance</i>	A FlexCAN instance number
<i>data</i>	The FlexCAN platform data
<i>enable_err_interrupts</i>	1 if enabled, 0 if not

Returns

0 if successful; non-zero failed

4.2.3.8 **uint32_t FLEXCAN_DRV_Deinit (uint8_t *instance*)**

Parameters

<i>instance</i>	A FlexCAN instance number
-----------------	---------------------------

Returns

0 if successful; non-zero failed

4.2.3.9 **flexcan_status_t FLEXCAN_DRV_ConfigTxMb (uint8_t *instance*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, flexcan_data_info_t * *tx_info*, uint32_t *msg_id*)**

FlexCAN Driver

Parameters

<i>instance</i>	A FlexCAN instance number
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>tx_info</i>	Data info
<i>msg_id</i>	ID of the message to transmit

Returns

0 if successful; non-zero failed

4.2.3.10 flexcan_status_t FLEXCAN_DRV_Send (uint8_t *instance*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, flexcan_data_info_t * *tx_info*, uint32_t *msg_id*, uint8_t * *mb_data*)

Parameters

<i>instance</i>	A FlexCAN instance number
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>tx_info</i>	Data info
<i>msg_id</i>	ID of the message to transmit
<i>mb_data</i>	Bytes of the FlexCAN message

Returns

0 if successful; non-zero failed

4.2.3.11 flexcan_status_t FLEXCAN_DRV_ConfigRxMb (uint8_t *instance*, const flexcan_user_config_t * *data*, uint32_t *mb_idx*, flexcan_data_info_t * *rx_info*, uint32_t *msg_id*)

Parameters

<i>instance</i>	A FlexCAN instance number
<i>data</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>rx_info</i>	Data info
<i>msg_id</i>	ID of the message to transmit

Returns

0 if successful; non-zero failed

4.2.3.12 **flexcan_status_t FLEXCAN_DRV_ConfigRxFifo (uint8_t *instance*, const flexcan_user_config_t * *data*, flexcan_rx_fifo_id_element_format_t *id_format*, flexcan_id_table_t * *id_filter_table*)**

Parameters

<i>instance</i>	A FlexCAN instance number
<i>data</i>	The FlexCAN platform data
<i>id_format</i>	The format of the Rx FIFO ID Filter Table Elements
<i>id_filter_table</i>	The ID filter table elements which contain RTR bit, IDE bit, and RX message ID

Returns

0 if successful; non-zero failed.

4.2.3.13 **flexcan_status_t FLEXCAN_DRV_RxMessageBuffer (uint8_t *instance*, const flexcan_user_config_t * *config*, uint32_t *mb_idx*, flexcan_mb_t * *data*)**

Parameters

<i>instance</i>	A FlexCAN instance number
-----------------	---------------------------

FlexCAN Driver

<i>config</i>	The FlexCAN platform data
<i>mb_idx</i>	Index of the message buffer
<i>data</i>	The FlexCAN receive message buffer data.

Returns

0 if successful; non-zero failed

4.2.3.14 **flexcan_status_t FLEXCAN_DRV_RxFifo (uint8_t *instance*, const flexcan_user_config_t * *config*, flexcan_mb_t * *data*)**

Parameters

<i>instance</i>	A FlexCAN instance number
<i>config</i>	The FlexCAN platform data
<i>data</i>	The FlexCAN receive message buffer data.

Returns

0 if successful; non-zero failed

4.2.3.15 **void FLEXCAN_DRV_IRQHandler (uint8_t *instance*)**

Parameters

<i>instance</i>	The FlexCAN instance number.
-----------------	------------------------------

Chapter 5

Comparator (CMP)

The Kinetis SDK provides both HAL and Peripheral drivers for the Comparator (CMP) block of Kinetis devices.

Modules

- [CMP HAL Driver](#)

This part describes the programming interface of the CMP HAL driver.

- [CMP Peripheral Driver](#)

This part describes the programming interface of the CMP Peripheral driver.

5.1 CMP HAL Driver

This chapter describes the programming interface of the CMP HAL driver.

Enumerations

- enum `cmp_status_t` {
 `kStatus_CMP_Success` = 0U,
 `kStatus_CMP_InvalidArgument` = 1U,
 `kStatus_CMP_Failed` = 2U }
CMP status return codes.
- enum `cmp_hysteresis_mode_t` {
 `kCmpHystersisOfLevel0` = 0U,
 `kCmpHystersisOfLevel1` = 1U,
 `kCmpHystersisOfLevel2` = 2U,
 `kCmpHystersisOfLevel3` = 3U }
Defines the hard block hysteresis control level selections.
- enum `cmp_filter_counter_mode_t` {
 `kCmpFilterCountSampleOf0` = 0U,
 `kCmpFilterCountSampleOf1` = 1U,
 `kCmpFilterCountSampleOf2` = 2U,
 `kCmpFilterCountSampleOf3` = 3U,
 `kCmpFilterCountSampleOf4` = 4U,
 `kCmpFilterCountSampleOf5` = 5U,
 `kCmpFilterCountSampleOf6` = 6U,
 `kCmpFilterCountSampleOf7` = 7U }
Defines the filter sample counter selections.
- enum `cmp_dac_ref_volt_src_mode_t` {
 `kCmpDacRefVoltSrcOf1` = 0U,
 `kCmpDacRefVoltSrcOf2` = 1U }
Defines the reference voltage source selections for the internal DAC.
- enum `cmp_chn_mux_mode_t` {
 `kCmpInputChn0` = 0U,
 `kCmpInputChn1` = 1U,
 `kCmpInputChn2` = 2U,
 `kCmpInputChn3` = 3U,
 `kCmpInputChn4` = 4U,
 `kCmpInputChn5` = 5U,
 `kCmpInputChn6` = 6U,
 `kCmpInputChn7` = 7U,
 `kCmpInputChnDac` = `kCmpInputChn7` }
Defines the CMP channel mux selection.

Functions

- void **CMP_HAL_Init** (uint32_t baseAddr)

Resets the CMP registers to a known state.
- static void **CMP_HAL_SetFilterCounterMode** (uint32_t baseAddr, cmp_filter_counter_mode_t mode)

Sets the filter sample count.
- static void **CMP_HAL_SetHysteresisMode** (uint32_t baseAddr, cmp_hysteresis_mode_t mode)

Sets the programmable hysteresis level.
- static void **CMP_HAL_Enable** (uint32_t baseAddr)

Enables the comparator in the CMP module.
- static void **CMP_HAL_Disable** (uint32_t baseAddr)

Disables the comparator in the CMP module.
- static void **CMP_HAL_SetOutputPinCmd** (uint32_t baseAddr, bool enable)

Switches to enable the compare output signal connecting to pin.
- static void **CMP_HAL_SetUnfilteredOutCmd** (uint32_t baseAddr, bool enable)

Switches to enable the filter for output of compare logic in CMP module.
- static void **CMP_HAL_SetInvertLogicCmd** (uint32_t baseAddr, bool enable)

Switches to enable the polarity of the analog comparator function.
- static void **CMP_HAL_SetHighSpeedCmd** (uint32_t baseAddr, bool enable)

Switches to enable the power (speed) comparison mode in CMP module.
- static void **CMP_HAL_SetSampleModeCmd** (uint32_t baseAddr, bool enable)

Switches to enable the sample mode in CMP module.
- static void **CMP_HAL_SetFilterPeriodValue** (uint32_t baseAddr, uint8_t value)

Sets the filter sample period in CMP module.
- static bool **CMP_HAL_GetOutputLogic** (uint32_t baseAddr)

Gets the comparator logic output in CMP module.
- static bool **CMP_HAL_GetOutputFallingFlag** (uint32_t baseAddr)

Gets the logic output falling edge event in the CMP module.
- static void **CMP_HAL_ClearOutputFallingFlag** (uint32_t baseAddr)

Clears the logic output falling edge event in the CMP module.
- static bool **CMP_HAL_GetOutputRisingFlag** (uint32_t baseAddr)

Gets the logic output rising edge event in the CMP module.
- static void **CMP_HAL_ClearOutputRisingFlag** (uint32_t baseAddr)

Clears the logic output rising edge event in the CMP module.
- static void **CMP_HAL_SetOutputFallingIntCmd** (uint32_t baseAddr, bool enable)

Switches to enable the requesting interrupt when the falling-edge on COUT has occurred.
- static bool **CMP_HAL_GetOutputFallingIntCmd** (uint32_t baseAddr)

Gets the interrupt request switcher on COUT falling-edge.
- static void **CMP_HAL_SetOutputRisingIntCmd** (uint32_t baseAddr, bool enable)

Gets the interrupt request switcher on COUT rising-edge.
- static bool **CMP_HAL_GetOutputRisingIntCmd** (uint32_t baseAddr)

Gets the interrupt request switcher on COUT rising-edge.
- static void **CMP_HAL_SetDacCmd** (uint32_t baseAddr, bool enable)

Switches to enable the internal 6-bit DAC in the CMP module.
- static void **CMP_HAL_SetDacRefVoltSrcMode** (uint32_t baseAddr, cmp_dac_ref_volt_src_mode_t mode)

Sets the reference voltage source for the internal 6-bit DAC in the CMP module.
- static void **CMP_HAL_SetDacValue** (uint32_t baseAddr, uint8_t value)

Sets the output value for the internal 6-bit DAC in the CMP module.
- static void **CMP_HAL_SetPlusInputChnMuxMode** (uint32_t baseAddr, cmp_chn_mux_mode_t mode)

Sets the plus input channel multiplexer mode for the internal 6-bit DAC in the CMP module.

CMP HAL Driver

t mode)

Sets the plus channel for the analog comparator.

- static void **CMP_HAL_SetMinusInputChnMuxMode** (uint32_t baseAddr, **cmp_chn_mux_mode_t** mode)

Sets the minus channel for the analog comparator.

5.1.0.16 CMP HAL Driver

Overview

The chapter describes the programming interface of the CMP HAL driver.

5.1.1 Enumeration Type Documentation

5.1.1.1 enum cmp_status_t

Enumerator

kStatus_CMP_Success Success.

kStatus_CMP_InvalidArgument Invalid argument existed.

kStatus_CMP_Failed Execution failed.

5.1.1.2 enum cmp_hysteresis_mode_t

The hysteresis control level indicates the smallest window between the two inputs when asserting the change of output. See the appropriate Data Sheet for detailed electrical characteristics. Generally, the lower level represents the smaller window.

Enumerator

kCmpHystersisOfLevel0 Level 0.

kCmpHystersisOfLevel1 Level 1.

kCmpHystersisOfLevel2 Level 2.

kCmpHystersisOfLevel3 Level 3.

5.1.1.3 enum cmp_filter_counter_mode_t

The selection item represents the number of consecutive samples that must agree prior to the comparator output filter accepting a new output state.

Enumerator

kCmpFilterCountSampleOf0 Disable the filter.

- kCmpFilterCountSampleOf1* One sample must agree.
- kCmpFilterCountSampleOf2* 2 consecutive samples must agree.
- kCmpFilterCountSampleOf3* 3 consecutive samples must agree.
- kCmpFilterCountSampleOf4* 4 consecutive samples must agree.
- kCmpFilterCountSampleOf5* 5 consecutive samples must agree.
- kCmpFilterCountSampleOf6* 6 consecutive samples must agree.
- kCmpFilterCountSampleOf7* 7 consecutive samples must agree.

5.1.1.4 enum cmp_dac_ref_volt_src_mode_t

Enumerator

- kCmpDacRefVoltSrcOf1* Vin1 - Vref_out.
- kCmpDacRefVoltSrcOf2* Vin2 - Vdd.

5.1.1.5 enum cmp_chn_mux_mode_t

Enumerator

- kCmpInputChn0* Comparator input channel 0.
- kCmpInputChn1* Comparator input channel 1.
- kCmpInputChn2* Comparator input channel 2.
- kCmpInputChn3* Comparator input channel 3.
- kCmpInputChn4* Comparator input channel 4.
- kCmpInputChn5* Comparator input channel 5.
- kCmpInputChn6* Comparator input channel 6.
- kCmpInputChn7* Comparator input channel 7.
- kCmpInputChnDac* Comparator input channel 7.

5.1.2 Function Documentation

5.1.2.1 void CMP_HAL_Init (uint32_t baseAddr)

This function resets the CMP's registers to a known state. This state is defined in Reference Manual, which is power on reset value.

Parameters

CMP HAL Driver

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

5.1.2.2 static void CMP_HAL_SetFilterCounterMode (uint32_t *baseAddr*, cmp_filter_counter_mode_t *mode*) [inline], [static]

This function sets the filter sample count. The value of the filter sample count represents the number of consecutive samples that must agree prior to the comparator output filter accepting a new output state.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Filter count value mode, see to "cmp_filter_counter_mode_t".

5.1.2.3 static void CMP_HAL_SetHysteresisMode (uint32_t *baseAddr*, cmp_hysteresis_mode_t *mode*) [inline], [static]

This function defines the programmable hysteresis level. The hysteresis values associated with each level are device-specific. See the Data Sheet of the device for the exact values. Also, see the "cmp_hysteresis_mode_t" for some additional information.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Hysteresis level, see to "cmp_hysteresis_mode_t".

5.1.2.4 static void CMP_HAL_Enable (uint32_t *baseAddr*) [inline], [static]

This function enables the comparator in the CMP module. The analog comparator is the core component in the CMP module. Only when it is enabled, all the other functions for advanced features are meaningful.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

5.1.2.5 static void CMP_HAL_Disable (uint32_t *baseAddr*) [inline], [static]

This function disables the comparator in CMP module. The analog comparator is the core component in CMP module. When it is disabled, it remains in the off state, and consumes no power.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

5.1.2.6 static void CMP_HAL_SetOutputPinCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the compare output signal connecting to pin. The comparator output (CMPO) is driven out on the associated CMPO output pin if the comparator owns the pin.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to enable the feature.

5.1.2.7 static void CMP_HAL_SetUnfilteredOutCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the filter for output of compare logic in CMP module. When enabled, it sets the unfiltered comparator output (CMPO) to equal COUT. When disabled, it sets the filtered comparator output(CMPO) to equal COUTA.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to enable the feature.

5.1.2.8 static void CMP_HAL_SetInvertLogicCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the polarity of the analog comparator function. When enabled, it inverts the comparator output logic.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

CMP HAL Driver

<i>enable</i>	Switcher to enable the feature.
---------------	---------------------------------

5.1.2.9 static void CMP_HAL_SetHighSpeedCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the power (speed) comparison mode in CMP module. When enabled, it selects the High-Speed (HS) comparison mode. In this mode, CMP has faster output propagation delay and higher current consumption.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to enable the feature.

5.1.2.10 static void CMP_HAL_SetSampleModeCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the sample mode in CMP module. When any sampled mode is active, COUTA is sampled whenever a rising-edge of filter block clock input or WINDOW/SAMPLE signal is detected.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to enable the feature.

5.1.2.11 static void CMP_HAL_SetFilterPeriodValue (uint32_t *baseAddr*, uint8_t *value*) [inline], [static]

This function sets the filter sample period in CMP module. The setting value specifies the sampling period, in bus clock cycles, of the comparator output filter, when sample mode is disabled. Setting the value to 0x0 disables the filter. This API has no effect when sample mode is enabled. In that case, the external SAMPLE signal is used to determine the sampling period.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>value</i>	Count of bus clock cycles for per sample.

5.1.2.12 static bool CMP_HAL_GetOutputLogic (uint32_t *baseAddr*) [inline], [static]

This function gets the comparator logic output in CMP module. It returns the current value of the analog comparator output. The value is reset to 0 and read as de-assert value when the CMP module is disabled. When setting to invert mode, the comparator logic output is also inverted.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

The logic output is assert or not.

5.1.2.13 static bool CMP_HAL_GetOutputFallingFlag (uint32_t *baseAddr*) [inline], [static]

This function gets the logic output falling edge event in the CMP module. It detects a falling-edge on COUT and returns the asserted state when the falling-edge on COUT has occurred.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

The falling-edge on COUT has occurred or not.

5.1.2.14 static void CMP_HAL_ClearOutputFallingFlag (uint32_t *baseAddr*) [inline], [static]

This function clears the logic output falling edge event in the CMP module.

CMP HAL Driver

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

5.1.2.15 static bool CMP_HAL_GetOutputRisingFlag (uint32_t *baseAddr*) [inline], [static]

This function gets the logic output rising edge event in the CMP module. It detects a rising-edge on COUT and returns the asserted state when the rising-edge on COUT has occurred.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

The rising-edge on COUT has occurred or not.

5.1.2.16 static void CMP_HAL_ClearOutputRisingFlag (uint32_t *baseAddr*) [inline], [static]

This function clears the logic output rising edge event in the CMP module.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

5.1.2.17 static void CMP_HAL_SetOutputFallingIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the requesting interrupt when the falling-edge on COUT has occurred. When enabled, it generates an interrupt request when falling-edge on COUT has occurred.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

<i>enable</i>	Switcher to enable the feature.
---------------	---------------------------------

**5.1.2.18 static bool CMP_HAL_GetOutputFallingIntCmd (uint32_t *baseAddr*)
[inline], [static]**

This function gets the switcher of the interrupt request on COUT falling-edge. When it is asserted, an interrupt request is generated when falling-edge on COUT has occurred.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

The status of switcher to enable the feature.

5.1.2.19 static void CMP_HAL_SetOutputRisingIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function gets the switcher of the interrupt request on COUT rising-edge. When it is asserted, an interrupt request is generated when the rising-edge on COUT has occurred.

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

The status of switcher to enable the feature.

**5.1.2.20 static bool CMP_HAL_GetOutputRisingIntCmd (uint32_t *baseAddr*)
[inline], [static]**

This function gets the switcher of the interrupt request on COUT rising-edge. When it is asserted, an interrupt request is generated when the rising-edge on COUT has occurred.

CMP HAL Driver

Parameters

<i>baseAddr</i>	Register base address for the module.
-----------------	---------------------------------------

Returns

The status of switcher to enable the feature.

5.1.2.21 static void CMP_HAL_SetDacCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the internal 6-bit DAC in the CMP module. When enabled, the internal 6-bit DAC can be used as an input channel to the analog comparator.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>enable</i>	Switcher to enable the feature.

5.1.2.22 static void CMP_HAL_SetDacRefVoltSrcMode (uint32_t *baseAddr*, cmp_dac_ref_volt_src_mode_t *mode*) [inline], [static]

This function sets the reference voltage source for the internal 6-bit DAC in the CMP module.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Selection of the feature, see to "cmp_dac_ref_volt_src_mode_t".

5.1.2.23 static void CMP_HAL_SetDacValue (uint32_t *baseAddr*, uint8_t *value*) [inline], [static]

This function sets the output value for the internal 6-bit DAC in the CMP module. The output voltage of DAC is $DACO = (V_{in}/64) * (value + 1)$ and the DACO range is from $V_{in}/64$ to V_{in} .

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>value</i>	Setting value, 6-bit available.

5.1.2.24 static void CMP_HAL_SetPlusInputChnMuxMode (uint32_t *baseAddr*, cmp_chn_mux_mode_t *mode*) [inline], [static]

This function sets the plus channel for the analog comparator. The plus and minus input channels come from the same channel mux. When an inappropriate operation selects the same input for both muxes, the comparator automatically shuts down to prevent becoming a noise generator. For channel use cases, see the appropriate data sheet for each SoC.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Channel mux mode, see to "cmp_chn_mux_mode_t".

5.1.2.25 static void CMP_HAL_SetMinusInputChnMuxMode (uint32_t *baseAddr*, cmp_chn_mux_mode_t *mode*) [inline], [static]

This function sets the minus channel for the analog comparator. The plus and minus input channels come from the same channel mux. When an inappropriate operation selects the same input for both muxes, the comparator automatically shuts down to prevent becoming a noise generator. For channel use cases, see the appropriate data sheet for each SoC.

Parameters

<i>baseAddr</i>	Register base address for the module.
<i>mode</i>	Channel mux mode, see to "cmp_chn_mux_mode_t".

CMP Peripheral Driver

5.2 CMP Peripheral Driver

This chapter describes the programming interface of the CMP Peripheral driver.

Data Structures

- struct `cmp_user_config_t`
Defines the structure to configure the comparator in the CMP module. [More...](#)
- struct `cmp_sample_filter_config_t`
Define structure of configuring Window/Filter in CMP module. [More...](#)
- struct `cmp_dac_config_t`
Define structure of configuring internal DAC in CMP module. [More...](#)
- struct `cmp_state_t`
Internal driver state information. [More...](#)

TypeDefs

- typedef void(* `cmp_callback_t`)(void)
Define type of user-defined callback function in CMP module.

Enumerations

- enum `cmp_sample_filter_mode_t` {
 `kCmpContinuousMode` = 0U,
 `kCmpSampleWithNoFilteredMode` = 1U,
 `kCmpSampleWithFilteredMode` = 2U,
 `kCmpWindowedMode` = 3U,
 `kCmpWindowedFilteredMode` = 4U }
Defines sample and filter mode selections in the CMP module.
- enum `cmp_flag_t` {
 `kCmpFlagOfCoutRising` = 0U,
 `kCmpFlagOfCoutFalling` = 1U }
Define type of flags for CMP event in CMP module.

Functions

- `cmp_status_t CMP_DRV_StructInitUserConfigDefault (cmp_user_config_t *userConfigPtr, cmp_chn_mux_mode_t plusInput, cmp_chn_mux_mode_t minusInput)`
Fill initial user configuration for default setting.
- `cmp_status_t CMP_DRV_Init (uint32_t instance, cmp_user_config_t *userConfigPtr, cmp_state_t *userStatePtr)`
Initialize the CMP module.
- `void CMP_DRV_Deinit (uint32_t instance)`
De-initialize the CMP module.
- `void CMP_DRV_Start (uint32_t instance)`

- Start the CMP module.
- void [CMP_DRV_Stop](#) (uint32_t instance)
 - Stop the CMP module.
- [cmp_status_t CMP_DRV_EnableDac](#) (uint32_t instance, [cmp_dac_config_t](#) *dacConfigPtr)
 - Enable the internal DAC in CMP module.
- void [CMP_DRV_DisableDac](#) (uint32_t instance)
 - Disable the internal DAC in CMP module.
- [cmp_status_t CMP_DRV_ConfigSampleFilter](#) (uint32_t instance, [cmp_sample_filter_config_t](#) *configPtr)
 - Configure the Sample feature in CMP module.
- bool [CMP_DRV_GetOutput](#) (uint32_t instance)
 - Get output of CMP module.
- bool [CMP_DRV_GetFlag](#) (uint32_t instance, [cmp_flag_t](#) flag)
 - Get state of CMP module.
- void [CMP_DRV_ClearFlag](#) (uint32_t instance, [cmp_flag_t](#) flag)
 - Clear event record of CMP module.
- [cmp_status_t CMP_DRV_InstallCallback](#) (uint32_t instance, [cmp_callback_t](#) userCallback)
 - Install the user-defined callback in CMP module.
- void [CMP_DRV_IRQHandler](#) (uint32_t instance)
 - Driver-defined ISR in CMP module.

5.2.0.26 CMP Peripheral Driver

Overview

The CMP peripheral driver configures the CMP (Comparator) and handles initialization and configuration of the CMP module.

Driver model building

CMP driver has three parts:

- Basic Comparator - This part handles the mechanism that compare the voltage from the two input channels and outputs the assertion if the plus side voltage is higher than the minus side voltage.
- Internal 6-bit DAC - The internal 6-bit DAC can be configured as one of the input channels to the basic comparator. It can provide a reference voltage level when compared with the other voltage signal.
- Sample/Filter - The Sample/Filter is an additional feature used to improve the signal of the comparator's output. The sample clock, the window mode and the accumulation filter can be used to configure the Sample/Filter. In fact, when configuring the Sample/Filter, limited settings are available. See the definition of the "cmp_sample_filter_mode_t" in the file "fsl_cmp_driver.h" or a reference manual for more information.

API functions for each part in the CMP driver module are separated and can be customized according to the application flexibility.

Call diagram

1. Make sure the pin mux settings for the CMP are ready before using the driver.
2. Call the `CMP_DRV_Init()` function to initialize the basic comparator. A configuration structure, `cmp_user_config_t`, is used to keep the initialization information. It can be filled by the application for a specific case, by the `CMP_DRV_StructInitUserConfigDefault` function to fill it with an available setting. A memory block is needed as a variable, `cmp_state_t`, to allocate and keep state with the `CMP_DRV_Init()` function.
3. Optionally, call the `CMP_DRV_EnableDac()` function to configure the internal 6-bit DAC if it is used as one of input channel. Use the configuration structure, `cmp_dac_config_t`, when calling this function.
4. Optionally, call the `CMP_DRV_ConfigSampleFilter()` function to configure the Sample/Filter. A configuration structure, `cmp_sample_filter_config_t`, is needed to keep the configuration.
5. Optionally, call the `CMP_DRV_InstallCallback()` function to install the user-defined callback function into the interrupt routine service. When the CMP is configured to enable the interrupt, the installed callback function will be called after an interrupt event occurs.
6. Finally, the CMP automatically responds to external events.

This is an example to initialize and configure the CMP driver for typical use cases.

```
/* cmp_test.c */

#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include "board.h"
#include "fsl_cmp_driver.h"
#include "fsl_device_registers.h"

#define TEST_CMP_INSTANCE      (0U)

cmp_state_t TestCmpStateStruct;

/* Normal Interrupt mode. */
volatile bool bRisingEvent = false;
volatile bool bFallingEvent = false;

static void CMP_TEST_ISR(void);
static void CMP_TEST_InitIO(void);

int main(void)
{
    cmp_user_config_t TestCmpUserConfigStruct;
    cmp_sample_filter_config_t TestCmpSampleFilterConfigStruct;
    cmp_dac_config_t TestCmpDacConfigStruct;

    hardware_init();
    dbg_uart_init();

    printf("CMP PD TEST: Start...\r\n");

    /* Init IO for CMP. */
    CMP_TEST_InitIO();

    /* Init the CMP comparator. */
    CMP_DRV_StructInitUserConfigDefault(&TestCmpUserConfigStruct,
                                         kCmpInputChn0, kCmpInputChnDac);
    TestCmpUserConfigStruct.risingIntEnable = true;
    TestCmpUserConfigStruct.fallingIntEnable = true;
```

```

    CMP_DRV_Init (TEST_CMP_INSTANCE, &TestCmpUserConfigStruct, &TestCmpStateStruct);

    /* Configure the internal DAC when in used. */
    TestCmpDacConfigStruct.dacValue = 32U; /* 0U - 63U */
    TestCmpDacConfigStruct.refVoltSrcMode = kCmpDacRefVoltSrcOf2;
    CMP_DRV_EnableDac (TEST_CMP_INSTANCE, &TestCmpDacConfigStruct);

    /* Configure the Sample/Filter Mode. */
    TestCmpSampleFilterConfigStruct.workMode = kCmpContinuousMode;
    CMP_DRV_ConfigSampleFilter (TEST_CMP_INSTANCE, &
        TestCmpSampleFilterConfigStruct);

    /* Start the CMP module. */
    CMP_DRV_Start (TEST_CMP_INSTANCE);

    /* Install the callback into interrupt. */
    CMP_DRV_InstallCallback (TEST_CMP_INSTANCE, CMP_TEST_ISR);

    while (1)
    {
        if (bRisingEvent)
        {
            printf("^\r\n");
            printf("  CMP output rising event occur!\r\n");
            printf("  CMP output level %d\r\n", CMP_DRV_GetOutput (TEST_CMP_INSTANCE) );
            bRisingEvent = false;
        }
        if (bFallingEvent)
        {
            printf("v\r\n");
            printf("  CMP output failing event occur!\r\n");
            printf("  CMP output level %d\r\n", CMP_DRV_GetOutput (TEST_CMP_INSTANCE) );
            bFallingEvent = false;
        }
    }

    //printf("CMP PD TEST: End.\r\n");
    //return 0;
}

static void CMP_TEST_ISR (void)
{
    if (CMP_DRV_GetFlag (TEST_CMP_INSTANCE, kCmpFlagOfCoutRising) )
    {
        if (!bRisingEvent)
        {
            bRisingEvent = true;
        }
    }
    if (CMP_DRV_GetFlag (TEST_CMP_INSTANCE, kCmpFlagOfCout Falling) )
    {
        if (!bFallingEvent)
        {
            bFallingEvent = true;
        }
    }
}

static void CMP_TEST_InitIO (void)
{
    /* TWR-K64F120M */
    PORTC->PCR[6] = PORT_PCR_MUX(0U); /* PTC6 - CMP0_IN0 */
    PORTC->PCR[7] = PORT_PCR_MUX(0U); /* PTC7 - CMP0_IN1 */
    PORTC->PCR[5] = PORT_PCR_MUX(6U); /* PTC5 - CMP0_OUT */
}

```

5.2.1 Data Structure Documentation

5.2.1.1 struct cmp_user_config_t

This type of structure keeps the configuration for the comparator inside the CMP module. With the configuration, the CMP can be set as a normal comparator without additional features.

Data Fields

- **cmp_hysteresis_mode_t hysteresisMode**
Set the hysteresis level.
- **bool pinoutEnable**
Enable to output the CMPO to pin.
- **bool pinoutUnfilteredEnable**
Enable to output unfiltered result to CMPO.
- **bool invertEnable**
Enable to invert the comparator's result.
- **bool highSpeedEnable**
Enable to work in speed mode.
- **bool risingIntEnable**
Enable to use CMPO rising interrupt.
- **bool fallingIntEnable**
Enable to use CMPO falling interrupt.
- **cmp_chn_mux_mode_t plusChnMux**
Set the Plus side input to comparator.
- **cmp_chn_mux_mode_t minusChnMux**
Set the Minus side input to comparator.

5.2.1.1.0.6 Field Documentation

5.2.1.1.0.6.1 `cmp_hysteresis_mode_t cmp_user_config_t::hysteresisMode`

5.2.1.1.0.6.2 `bool cmp_user_config_t::pinoutEnable`

5.2.1.1.0.6.3 `bool cmp_user_config_t::pinoutUnfilteredEnable`

5.2.1.1.0.6.4 `bool cmp_user_config_t::invertEnable`

5.2.1.1.0.6.5 `bool cmp_user_config_t::highSpeedEnable`

5.2.1.1.0.6.6 `bool cmp_user_config_t::risingIntEnable`

5.2.1.1.0.6.7 `bool cmp_user_config_t::fallingIntEnable`

5.2.1.1.0.6.8 `cmp_chn_mux_mode_t cmp_user_config_t::plusChnMux`

5.2.1.1.0.6.9 `cmp_chn_mux_mode_t cmp_user_config_t::minusChnMux`

5.2.1.2 `struct cmp_sample_filter_config_t`

This type of structure is to keep the configuration for Window/Filter inside the CMP module. With the configuration, the CMP module can work in some advanced mode.

Data Fields

- `cmp_sample_filter_mode_t workMode`
Sample/Filter's work mode.
- `bool useExtSampleOrWindow`
Switcher to use external WINDOW/SAMPLE signal.
- `uint8_t filterClkDiv`
Filter's prescaler which divides from the bus clock.
- `cmp_filter_counter_mode_t filterCount`
Sample count for filter, see to "cmp_filter_counter_mode_t".

5.2.1.2.0.7 Field Documentation

5.2.1.2.0.7.1 `cmp_sample_filter_mode_t cmp_sample_filter_config_t::workMode`

5.2.1.2.0.7.2 `bool cmp_sample_filter_config_t::useExtSampleOrWindow`

5.2.1.2.0.7.3 `uint8_t cmp_sample_filter_config_t::filterClkDiv`

5.2.1.2.0.7.4 `cmp_filter_counter_mode_t cmp_sample_filter_config_t::filterCount`

5.2.1.3 `struct cmp_dac_config_t`

This type of structure is to keep the configuration for DAC inside the CMP module. With the configuration, the internal DAC would provide a reference voltage level, it is chosen as the input of CMP.

Data Fields

- `cmp_dac_ref_volt_src_mode_t refVoltSrcMode`
Select the reference voltage source for internal DAC.
- `uint8_t dacValue`
Set the value for internal DAC.

5.2.1.3.0.8 Field Documentation

5.2.1.3.0.8.1 `cmp_dac_ref_volt_src_mode_t cmp_dac_config_t::refVoltSrcMode`

5.2.1.3.0.8.2 `uint8_t cmp_dac_config_t::dacValue`

5.2.1.4 `struct cmp_state_t`

The contents of this structure are internal to the driver and should not be modified by users. Also, contents of the structure are subject to change in future releases.

5.2.2 Enumeration Type Documentation

5.2.2.1 `enum cmp_sample_filter_mode_t`

The comparator sample/filter is available in several modes. Use the enumeration to identify the comparator's condition:

`kCmpContinuousMode` - Continuous Mode: Both window control and filter blocks are completely bypassed. The output of the comparator is updated continuously. `kCmpSampleWithNoFilteredMode` - Sample, Non-Filtered Mode: Window control is completely bypassed. The output of the comparator is sampled whenever a rising-edge is detected on the filter block clock input. The filter clock prescaler can be configured as the divider from the bus clock. `kCmpSampleWithFilteredMode` - Sample, Filtered Mode: Similar to "Sample, Non-Filtered Mode", but the filter is active in this mode. The filter counter value becomes configurable as well. `kCmpWindowedMode` - Windowed Mode: In Window Mode, only the output of the analog comparator is passed when the Window signal is high. The last latched value is held when the Window signal is low. `kCmpWindowedFilteredMode` - Window/Filtered Mode: This mode complex and uses both window and filtering features. It also has the highest latency of all modes. This can be approximated as follows: up to 1 bus clock synchronization in the window function

- ((filter counter * filter prescaler) + 1) bus clock for the filter function.

Enumerator

`kCmpContinuousMode` Continuous Mode.

`kCmpSampleWithNoFilteredMode` Sample, Non-Filtered Mode.

`kCmpSampleWithFilteredMode` Sample, Filtered Mode.

`kCmpWindowedMode` Windowed Mode.

`kCmpWindowedFilteredMode` Window/Filtered Mode.

5.2.2.2 enum cmp_flag_t

Enumerator

kCmpFlagOfCoutRising Identifier to indicate if the COUT change from logic zero to one.

kCmpFlagOfCoutFalling Identifier to indicate if the COUT change from logic one to zero.

5.2.3 Function Documentation

5.2.3.1 cmp_status_t CMP_DRV_StructInitUserConfigDefault (cmp_user_config_t * *userConfigPtr*, cmp_chn_mux_mode_t *plusInput*, cmp_chn_mux_mode_t *minusInput*)

This function is to fill initial user configuration for default setting. The default setting will make the CMP module at least to be an comparator. It includes the setting of : .hysteresisMode = kCmpHystersisOfLevel0 .pinoutEnable = true .pinoutUnfilteredEnable = true .invertEnable = false .highSpeedEnable = false .dmaEnable = false .risingIntEnable = false .fallingIntEnable = false .triggerEnable = false However, it is still recommended to fill some fields of structure such as channel mux according to application. Note that this API will not set the configuration to hardware.

Parameters

<i>userConfigPtr</i>	Pointer to structure of configuration, see to "cmp_user_config_t".
<i>plusInput</i>	Plus Input mux selection, see to "cmp_chn_mux_mode_t".
<i>minusInput</i>	Minus Input mux selection, see to "cmp_chn_mux_mode_t".

Returns

Execution status.

5.2.3.2 cmp_status_t CMP_DRV_Init (uint32_t *instance*, cmp_user_config_t * *userConfigPtr*, cmp_state_t * *userStatePtr*)

This function is to initialize the CMP module. It will enable the clock and set the interrupt switcher for CMP module. And the CMP module will be configured for the basic comparator.

Parameters

CMP Peripheral Driver

<i>instance</i>	CMP instance id.
<i>userConfigPtr</i>	Pointer to structure of configuration, see to "cmp_user_config_t".
<i>userStatePtr</i>	Pointer to structure of context, see to "cmp_state_t".

Returns

Execution status.

5.2.3.3 void CMP_DRV_Deinit (uint32_t *instance*)

This function is to de-initialize the CMP module. It will shutdown the CMP's clock and disable the interrupt. This API should be called when CMP is no longer used in application and it will help to reduce the power consumption.

Parameters

<i>instance</i>	CMP instance id.
-----------------	------------------

5.2.3.4 void CMP_DRV_Start (uint32_t *instance*)

This function is to start the CMP module. The configuration would not take effect until the module is started.

Parameters

<i>instance</i>	CMP instance id.
-----------------	------------------

5.2.3.5 void CMP_DRV_Stop (uint32_t *instance*)

This function is to stop the CMP module. Note that this API would shutdown the module, but only pauses the features tenderly.

Parameters

<i>instance</i>	CMP instance id.
-----------------	------------------

5.2.3.6 cmp_status_t CMP_DRV_EnableDac (uint32_t *instance*, cmp_dac_config_t * *dacConfigPtr*)

This function is to enable the internal DAC in CMP module. It will take effect actually only when internal DAC has been chosen as one of input channel for comparator. Then the DAC channel can be programmed

to provide a reference voltage level.

CMP Peripheral Driver

Parameters

<i>instance</i>	CMP instance id.
<i>dacConfigPtr</i>	Pointer to structure of configuration, see to "cmp_dac_config_t".

Returns

Execution status.

5.2.3.7 void CMP_DRV_DisableDac (uint32_t *instance*)

This function is to disable the internal DAC in CMP module. It should be called if the internal DAC is no longer used in application.

Parameters

<i>instance</i>	CMP instance id.
-----------------	------------------

5.2.3.8 cmp_status_t CMP_DRV_ConfigSampleFilter (uint32_t *instance*, cmp_sample_filter_config_t * *configPtr*)

This function is to configure the CMP working in Sample modes. These modes are some advanced features beside the basic comparator. They may be about Windowed Mode, Filter Mode and so on. See to "cmp_sample_filter_config_t" for detailed description.

Parameters

<i>instance</i>	CMP instance id.
<i>cmp_sample_filter_config_t</i>	Pointer to structure of configuration. see to "cmp_sample_filter_config_t".

Returns

Execution status.

5.2.3.9 bool CMP_DRV_GetOutput (uint32_t *instance*)

This function is to get the output of CMP module. The output source depends on the configuration when initializing the comparator. When *cmp_user_config_t.pinoutUnfilteredEnable* = false, the output will be processed by filter. Otherwise, the output would be the signal did not pass the filter.

Parameters

<i>instance</i>	CMP instance id.
-----------------	------------------

Returns

Output logic's assertion. When no invert, plus side > minus side, it will be true.

5.2.3.10 **bool CMP_DRV_GetFlag (uint32_t *instance*, cmp_flag_t *flag*)**

This function is to get the state of CMP module. It will return if indicated event has been detected.

Parameters

<i>instance</i>	CMP instance id.
<i>flag</i>	Represent events or states, see to "cmp_flag_t".

Returns

Assertion if indicated event occurs.

5.2.3.11 **void CMP_DRV_ClearFlag (uint32_t *instance*, cmp_flag_t *flag*)**

This function is to clear event record of CMP module.

Parameters

<i>instance</i>	CMP instance id.
<i>flag</i>	Represent events or states, see to "cmp_flag_t".

5.2.3.12 **cmp_status_t CMP_DRV_InstallCallback (uint32_t *instance*, cmp_callback_t *userCallback*)**

This function is to install the user-defined callback in CMP module. When an CMP interrupt request is served, the callback will be executed inside the ISR.

CMP Peripheral Driver

Parameters

<i>instance</i>	CMP instance id.
<i>userCallback</i>	User-defined callback function.

5.2.3.13 void CMP_DRV_IRQHandler (uint32_t *instance*)

This function is the driver-defined ISR in CMP module. It includes the process for interrupt mode defined by driver. Currently, it will be called inside the system-defined ISR.

Parameters

<i>instance</i>	CMP instance id.
-----------------	------------------

Chapter 6

Digital-to-Analog Converter (DAC)

The Kinetis SDK provides both HAL and Peripheral drivers for the Digital-to-Analog Converter block of Kinetis devices.

Modules

- [DAC HAL Driver](#)

This part describes the programming interface of the DAC HAL driver.

- [DAC Peripheral Driver](#)

This part describes the programming interface of the DAC Peripheral driver.

DAC HAL Driver

6.1 DAC HAL Driver

This chapter describes the programming interface of the DAC HAL driver.

Enumerations

- enum `dac_status_t` {
 `kStatus_DAC_Success` = 0U,
 `kStatus_DAC_InvalidArgument` = 1U,
 `kStatus_DAC_Failed` = 2U }
 DAC status return codes.
- enum `dac_ref_volt_src_mode_t` {
 `kDacRefVoltSrcOfVref1` = 0U,
 `kDacRefVoltSrcOfVref2` = 1U }
 Defines the type of selection for DAC module's reference voltage source.
- enum `dac_trigger_mode_t` {
 `kDacTriggerByHardware` = 0U,
 `kDacTriggerBySoftware` = 1U }
 Defines the type of selection for DAC module trigger mode.
- enum `dac_buff_watermark_mode_t` {
 `kDacBuffWatermarkFromUpperAs1Word` = 0U,
 `kDacBuffWatermarkFromUpperAs2Word` = 1U,
 `kDacBuffWatermarkFromUpperAs3Word` = 2U,
 `kDacBuffWatermarkFromUpperAs4Word` = 3U }
 Defines the type of selection for buffer watermark mode.
- enum `dac_buff_work_mode_t` {
 `kDacBuffWorkAsNormalMode` = 0U,
 `kDacBuffWorkAsSwingMode` = 1U,
 `kDacBuffWorkAsOneTimeScanMode` = 2U,
 `kDacBuffWorkAsFIFOMode` = 3U }
 Defines the type of selection for buffer work mode.

Functions

- void `DAC_HAL_Init` (uint32_t baseAddr)
 Resets all configurable registers to be in the reset state for DAC.
- void `DAC_HAL_SetBuffValue` (uint32_t baseAddr, uint8_t index, uint16_t value)
 Sets the 12-bit value for the DAC items in the buffer.
- uint16_t `DAC_HAL_GetBuffValue` (uint32_t baseAddr, uint8_t index)
 Gets the 12-bit value from the DAC item in the buffer.
- static void `DAC_HAL_ClearBuffIndexUpperFlag` (uint32_t baseAddr)
 Clears the flag of the DAC buffer read pointer.
- static bool `DAC_HAL_GetBuffIndexUpperFlag` (uint32_t baseAddr)
 Gets the flag of DAC buffer read pointer when it hits the bottom position.
- static void `DAC_HAL_ClearBuffIndexStartFlag` (uint32_t baseAddr)
 Clears the flag of the DAC buffer read pointer when it hits the top position.
- static bool `DAC_HAL_GetBuffIndexStartFlag` (uint32_t baseAddr)

- Gets the flag of the DAC buffer read pointer when it hits the top position.
- static void **DAC_HAL_Enable** (uint32_t baseAddr)
 - Enables the Programmable Reference Generator.
- static void **DAC_HAL_Disable** (uint32_t baseAddr)
 - Disables the Programmable Reference Generator.
- static void **DAC_HAL_SetRefVoltSrcMode** (uint32_t baseAddr, **dac_ref_volt_src_mode_t** mode)
 - Sets the reference voltage source mode for the DAC module.
- static void **DAC_HAL_SetTriggerMode** (uint32_t baseAddr, **dac_trigger_mode_t** mode)
 - Sets the trigger mode for the DAC module.
- static void **DAC_HAL_SetSoftTriggerCmd** (uint32_t baseAddr)
 - Triggers the converter with software.
- static void **DAC_HAL_SetLowPowerCmd** (uint32_t baseAddr, bool enable)
 - Switches to enable working in low power mode for the DAC module.
- static void **DAC_HAL_SetBuffIndexStartIntCmd** (uint32_t baseAddr, bool enable)
 - Switches to enable the interrupt when the buffer read pointer hits the top position.
- static void **DAC_HAL_SetBuffIndexUpperIntCmd** (uint32_t baseAddr, bool enable)
 - Switches to enable the interrupt when the buffer read pointer hits the bottom position.
- static void **DAC_HAL_SetDmaCmd** (uint32_t baseAddr, bool enable)
 - Switches to enable the DMA for DAC.
- static void **DAC_HAL_SetBuffWorkMode** (uint32_t baseAddr, **dac_buff_work_mode_t** mode)
 - Sets the work mode of the buffer for the DAC module.
- static void **DAC_HAL_SetBuffCmd** (uint32_t baseAddr, bool enable)
 - Switches to enable the buffer for the DAC module.
- static uint8_t **DAC_HAL_GetBuffUpperIndex** (uint32_t baseAddr)
 - Gets the buffer index upper limitation for the DAC module.
- static void **DAC_HAL_SetBuffUpperIndex** (uint32_t baseAddr, uint8_t index)
 - Sets the buffer index upper limitation for the DAC module.
- static uint8_t **DAC_HAL_GetBuffCurrentIndex** (uint32_t baseAddr)
 - Gets the current buffer index upper limitation for the DAC module.
- static void **DAC_HAL_SetBuffcurrentIndex** (uint32_t baseAddr, uint8_t index)
 - Sets the buffer index for the DAC module.

6.1.0.14 DAC HAL Driver

Overview

The DAC HAL driver masks the hardware and provides a comprehensible way to use the DAC hardware.

6.1.1 Enumeration Type Documentation

6.1.1.1 enum dac_status_t

Enumerator

kStatus_DAC_Success Success.

kStatus_DAC_InvalidArgument Invalid argument existed.

kStatus_DAC_Failed Execution failed.

DAC HAL Driver

6.1.1.2 enum dac_ref_volt_src_mode_t

See the appropriate SoC Reference Manual for actual connections.

Enumerator

- kDacRefVoltSrcOfVref1* Select DACREF_1 as the reference voltage.
- kDacRefVoltSrcOfVref2* Select DACREF_2 as the reference voltage.

6.1.1.3 enum dac_trigger_mode_t

Enumerator

- kDacTriggerByHardware* Select hardware trigger.
- kDacTriggerBySoftware* Select software trigger.

6.1.1.4 enum dac_buff_watermark_mode_t

If the buffer feature for DAC module is enabled, a watermark event will occur when the buffer index hits the watermark.

Enumerator

- kDacBuffWatermarkFromUpperAs1Word* Select 1 word away from the upper of buffer.
- kDacBuffWatermarkFromUpperAs2Word* Select 2 word away from the upper of buffer.
- kDacBuffWatermarkFromUpperAs3Word* Select 3 word away from the upper of buffer.
- kDacBuffWatermarkFromUpperAs4Word* Select 4 word away from the upper of buffer.

6.1.1.5 enum dac_buff_work_mode_t

There are three kinds of work modes when the DAC buffer is enabled. Normal mode - When the buffer index hits the upper level, it starts (0) on the next trigger. Swing mode - When the buffer index hits the upper level, it goes backward to the start and is reduced one-by-one on the next trigger. When the buffer index hits the start, it goes backward to the upper level and increases one-by-one on the next trigger. One-Time-Scan mode - The buffer index can only be increased on the next trigger. When the buffer index hits the upper level, it is not updated by the trigger. FIFO mode

Enumerator

- kDacBuffWorkAsNormalMode* Buffer works as Normal.
- kDacBuffWorkAsSwingMode* Buffer works as swing.
- kDacBuffWorkAsOneTimeScanMode* Buffer works as one time scan.
- kDacBuffWorkAsFIFOMode* Buffer works as FIFO.

6.1.2 Function Documentation

6.1.2.1 void DAC_HAL_Init(uint32_t *baseAddr*)

This function resets all configurable registers to be in the reset state for DAC. It should be called before configuring the DAC module.

DAC HAL Driver

Parameters

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

6.1.2.2 void DAC_HAL_SetBuffValue (*uint32_t baseAddr, uint8_t index, uint16_t value*)

This function sets the value assembled by the low 8 bits and high 4 bits of 12-bit DAC item in the buffer.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>index</i>	Buffer index.
<i>value</i>	Setting value.

6.1.2.3 uint16_t DAC_HAL_GetBuffValue (*uint32_t baseAddr, uint8_t index*)

This function gets the value assembled by the low 8 bits and high 4 bits of 12-bit DAC item in the buffer.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>index</i>	Buffer index.

Returns

Current setting value.

6.1.2.4 static void DAC_HAL_ClearBuffIndexUpperFlag (*uint32_t baseAddr*) [inline], [static]

This function clears the flag of the DAC buffer read pointer when it hits the bottom position.

Parameters

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

6.1.2.5 static bool DAC_HAL_GetBuffIndexUpperFlag (*uint32_t baseAddr*) [inline], [static]

This function gets the flag of DAC buffer read pointer when it hits the bottom position.

Parameters

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

Returns

Assertion of indicated event.

6.1.2.6 static void DAC_HAL_ClearBuffIndexStartFlag (uint32_t *baseAddr*) [inline], [static]

This function clears the flag of the DAC buffer read pointer when it hits the top position.

Parameters

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

6.1.2.7 static bool DAC_HAL_GetBuffIndexStartFlag (uint32_t *baseAddr*) [inline], [static]

This function gets the flag of the DAC buffer read pointer when it hits the top position.

Parameters

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

Returns

Assertion of indicated event.

6.1.2.8 static void DAC_HAL_Enable (uint32_t *baseAddr*) [inline], [static]

This function enables the Programmable Reference Generator. Then the DAC system is enabled.

Parameters

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

6.1.2.9 static void DAC_HAL_Disable (uint32_t *baseAddr*) [inline], [static]

This function disables the Programmable Reference Generator. Then the DAC system is disabled.

DAC HAL Driver

Parameters

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

6.1.2.10 static void DAC_HAL_SetRefVoltSrcMode (uint32_t *baseAddr*, dac_ref_volt_src_mode_t *mode*) [inline], [static]

This function sets the reference voltage source mode for the DAC module.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>mode</i>	Selection of enumeration mode. See to "dac_ref_volt_src_mode_t".

6.1.2.11 static void DAC_HAL_SetTriggerMode (uint32_t *baseAddr*, dac_trigger_mode_t *mode*) [inline], [static]

This function sets the trigger mode for the DAC module.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>mode</i>	Selection of enumeration mode. See to "dac_trigger_mode_t".

6.1.2.12 static void DAC_HAL_SetSoftTriggerCmd (uint32_t *baseAddr*) [inline], [static]

This function triggers the converter with software. If the DAC software trigger is selected and buffer enabled, calling this API advances the buffer read pointer once.

Parameters

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

6.1.2.13 static void DAC_HAL_SetLowPowerCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable working in low power mode for the DAC module.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>enable</i>	Switcher to assert the feature.

6.1.2.14 static void DAC_HAL_SetBuffIndexStartIntCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the interrupt when the buffer read pointer hits the top position.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>enable</i>	Switcher to assert the feature.

6.1.2.15 static void DAC_HAL_SetBuffIndexUpperIntCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the interrupt when the buffer read pointer hits the bottom position.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>enable</i>	Switcher to assert the feature.

6.1.2.16 static void DAC_HAL_SetDmaCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function switches to enable the DMA for the DAC module. When the DMA is enabled, DMA request is generated by the original interrupts, which are not presented on this module at the same time.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>enable</i>	Switcher to assert the feature.

6.1.2.17 static void DAC_HAL_SetBuffWorkMode (*uint32_t baseAddr, dac_buff_work_mode_t mode*) [inline], [static]

This function sets the work mode of the buffer for the DAC module.

DAC HAL Driver

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>mode</i>	Selection of enumeration mode. See to "dac_buff_work_mode_t".

6.1.2.18 static void DAC_HAL_SetBuffCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function switches to enable the buffer for the DAC module.

Parameters

<i>baseAddr</i>	The DAC peripheral base address.
<i>enable</i>	Switcher to assert the feature.

6.1.2.19 static uint8_t DAC_HAL_GetBuffUpperIndex (uint32_t *baseAddr*) [inline], [static]

This function gets the upper buffer index upper limitation for the DAC module.

Parameters

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

Returns

Value of buffer index upper limitation.

6.1.2.20 static void DAC_HAL_SetBuffUpperIndex (uint32_t *baseAddr*, uint8_t *index*) [inline], [static]

This function sets the upper buffer index upper limitation for the DAC module.

Parameters

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

<i>index</i>	Setting value of upper limitation for buffer index.
--------------	---

6.1.2.21 static uint8_t DAC_HAL_GetBuffCurrentIndex (uint32_t *baseAddr*) [inline], [static]

This function gets the current buffer index for the DAC module.

Parameters

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

Returns

Value of current buffer index.

6.1.2.22 static void DAC_HAL_SetBuffCurrentIndex (uint32_t *baseAddr*, uint8_t *index*) [inline], [static]

This function sets the upper buffer index for the DAC module.

Parameters

<i>baseAddr</i>	the DAC peripheral base address.
<i>index</i>	Setting value for buffer index.

DAC Peripheral Driver

6.2 DAC Peripheral Driver

This chapter describes the programming interface of the DAC Peripheral driver.

Data Structures

- struct `dac_buff_config_t`
Defines the configuration buffer structure inside the DAC module. [More...](#)
- struct `dac_user_config_t`
Defines the converted configuration structure. [More...](#)
- struct `dac_state_t`
Internal driver state information. [More...](#)

Typedefs

- `typedef void(* dac_callback_t)(void)`
Defines the type of the user-defined callback function.

Enumerations

- enum `dac_flag_t` {
 `kDacBuffIndexStartFlag` = 1U,
 `kDacBuffIndexUpperFlag` = 2U }
Defines the type of event flags.

Functions

- `dac_status_t DAC_DRV_StructInitUserConfigNormal (dac_user_config_t *userConfigPtr)`
Fills the initial user configuration for the DAC module without the interrupt and the buffer.
- `dac_status_t DAC_DRV_Init (uint32_t instance, dac_user_config_t *userConfigPtr)`
Initializes the converter.
- `void DAC_DRV_Deinit (uint32_t instance)`
De-initializes the DAC module converter.
- `void DAC_DRV_Output (uint32_t instance, uint16_t value)`
Drives the converter to output the DAC value.
- `dac_status_t DAC_DRV_EnableBuff (uint32_t instance, dac_buff_config_t *buffConfigPtr, dac_state_t *userStatePtr)`
Configures the internal buffer.
- `void DAC_DRV_DisableBuff (uint32_t instance)`
Disables the internal buffer.
- `dac_status_t DAC_DRV_SetBuffValue (uint32_t instance, uint8_t start, uint8_t offset, uint16_t arr[])`
Sets values into the DAC internal buffer.
- `uint16_t DAC_DRV_SoftTriggerBuff (uint32_t instance)`
Triggers the buffer by software and returns the current value.
- `uint8_t DAC_DRV_GetBufferIndex (uint32_t instance)`

- Gets the DAC buffer current index.
- void [DAC_DRV_ClearFlag](#) (uint32_t instance, [dac_flag_t](#) flag)
Clears the flag for an indicated event causing an interrupt.
- bool [DAC_DRV_GetFlag](#) (uint32_t instance, [dac_flag_t](#) flag)
Gets the flag for an indicated event causing an interrupt.
- [dac_status_t](#) [DAC_DRV_InstallCallback](#) (uint32_t instance, [dac_callback_t](#) userCallback)
Install the user-defined callback.
- void [DAC_DRV_IRQHandler](#) (uint32_t instance)
Driver-defined ISR in DAC module.

6.2.0.23 DAC Peripheral Driver

Overview

The DAC peripheral driver configures the DAC (Digital-to-Analog Converter). It also handles the module initialization and configuration by converting attributes.

Initialization

To initialize the DAC module, call the [DAC_DRV_Init\(\)](#) function and pass the configuration data structure, which can be filled by the [DAC_DRV_StructInitUserConfigNormal\(\)](#) function with the default settings for the converter. After it is initialized, the DAC module can function as a DAC converter.

The DAC module provides advanced features internally with the hardware buffer. To use the advanced features, the API of the [DAC_DRV_EnableBuff\(\)](#) function should be called to initialize the buffer.

Model building

The DAC module provides advanced features with the hardware DAC buffer.

When the DAC is enabled and the buffer is not enabled, the DAC module always converts the data in DAT0, the first item in the buffer, to analog output voltage. If the buffer is enabled, the DAC converts the items in the data buffer to analog output voltage according to the buffer configuration. The data buffer read pointer advances to the next word whenever any hardware or software trigger event occurs. The data buffer can be configured to operate in FIFO mode, Swing mode, or One-Time Scan mode:

- FIFO mode is the default mode for the buffer. The buffer works as a FIFO buffer. The read pointer increases once triggered. When the read pointer reaches the upper limit, it goes to zero during the next trigger event.
- Swing mode is similar to the FIFO mode. However, when the read pointer reaches the upper limit, it does not go to zero. It will descend by 1 in the next trigger events until reach to zero, and then turns back.
- One-time Scan mode. In One-time Scan mode, the read pointer increases by one once triggered. When it reaches the upper limit, it stops there. If read pointer is reset to the address other than the upper limit, it increases to the upper address and stops. If the software sets the read pointer to the

DAC Peripheral Driver

upper limit, the read pointer does not advance in this mode. When the buffer operation is switched from one mode to another, the read pointer does not change.

Call diagram

Four kinds of typical use cases are designed for the DAC module:

- Normal converter mode. Normal converter mode is working without the buffer and the trigger. It is the simplest way to use the DAC module.
- FIFO buffer mode. FIFO buffer mode enables the internal buffer and sets it as a FIFO mode.
- Swing buffer mode. Swing buffer mode enables the internal buffer and set it as a Swing mode.
- One-time Scan buffer mode. One-time Scan buffer mode enables the internal buffer and sets it as a One-time Scan mode.

The three use cases can function on the software trigger or hardware trigger. To use the hardware trigger, enable the hardware trigger setting in the DAC module. Then configure the other module that can generate the trigger, such as the PDB.

These are examples to initialize and configure the PDB driver for typical use cases:

Normal converter mode:

```
/* dac_test_normal.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_dac_driver.h"
#include "fsl_os_abstraction.h"

void DAC_TEST_NormalMode(uint32_t instance, uint16_t *buffPtr, uint8_t buffLen)
{
    dac_user_config_t MyDacUserConfigStruct;
    uint8_t i;

    /* Fill the structure with configuration of software trigger. */
    DAC_DRV_StructInitUserConfigNormal(&MyDacUserConfigStruct);

    /* Initialize the DAC Converter. */
    DAC_DRV_Init(instance, &MyDacUserConfigStruct);

    /* Output the DAC value. */
    for (i = 0U; i < buffLen; i++)
    {
        printf("DAC_DRV_Output: %d\r\n", buffPtr[i]);
        DAC_DRV_Output(instance, buffPtr[i]);
        OSA_TimeDelay(200);
    }

    /* De-initialize the DAC converter. */
    DAC_DRV_Deinit(instance);
}
```

FIFO buffer mode:

```
/* dac_test_buffer_fifo.c */
```

```

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_dac_driver.h"
#include "fsl_os_abstraction.h"

static dac_state_t MyDacStateStructForBufferFIFO;

extern void DAC_ISR_Buffer(void);

void DAC_TEST_BufferFIFOMode(uint32_t instance, uint16_t *buffPtr, uint8_t buffLen)
{
    dac_user_config_t MyDacUserConfigStruct;
    dac_buff_config_t MyDacBuffConfigStruct;
    uint8_t i;
    volatile uint16_t dacValue;

    /* Fill the structure with configuration of software trigger. */
    DAC_DRV_StructInitUserConfigNormal(&MyDacUserConfigStruct);

    /* Initialize the DAC Converter. */
    DAC_DRV_Init(instance, &MyDacUserConfigStruct);

    /* Enable the feature of DAC internal buffer. */
    MyDacBuffConfigStruct.bufIndexWatermarkIntEnable = true;
    MyDacBuffConfigStruct.bufIndexStartIntEnable = true;
    MyDacBuffConfigStruct.bufIndexUpperIntEnable = true;
    MyDacBuffConfigStruct.dmaEnable = false;
    MyDacBuffConfigStruct.watermarkMode = kDacBuffWatermarkFromUpperAs2Word
    ;
    MyDacBuffConfigStruct.bufWorkMode = kDacBuffWorkAsNormalMode;
    MyDacBuffConfigStruct.bufUpperIndex = buffLen - 1;
    DAC_DRV_EnableBuff(instance, &MyDacBuffConfigStruct, &MyDacStateStructForBufferFIFO);

    /* Fill the buffer with setting data. */
    DAC_DRV_SetBuffValue(instance, 0U, buffLen, buffPtr);

    /* Register the callback function for DAC buffer event. */
    DAC_DRV_InstallCallback(instance, DAC_ISR_Buffer);

    /* Trigger the buffer to output setting value. */
    for (i = 0U; i < buffLen*4U; i++)
    {
        dacValue = DAC_DRV_SoftTriggerBuff(instance);
        dacValue = dacValue;
        printf("DAC_DRV_SoftTriggerBuff: %d\r\n", dacValue);
        OSA_TimeDelay(200);
    }

    /* Disable the feature of DAC internal buffer. */
    DAC_DRV_DisableBuff(instance);

    /* De-initialize the DAC converter. */
    DAC_DRV_Deinit(instance);
}

```

Swing buffer mode:

```

/* dac_test_buffer_swing.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_dac_driver.h"
#include "fsl_os_abstraction.h"

```

DAC Peripheral Driver

```
static dac_state_t MyDacStateStructForBufferSwing;

extern void DAC_ISR_Buffer(void);

void DAC_TEST_BufferSwingMode(uint32_t instance, uint16_t *buffPtr, uint8_t buffLen)
{
    dac_user_config_t MyDacUserConfigStruct;
    dac_buff_config_t MyDacBuffConfigStruct;
    uint8_t i;
    volatile uint16_t dacValue;

    /* Fill the structure with configuration of software trigger. */
    DAC_DRV_StructInitUserConfigNormal(&MyDacUserConfigStruct);

    /* Initialize the DAC Converter. */
    DAC_DRV_Init(instance, &MyDacUserConfigStruct);

    /* Enable the feature of DAC internal buffer. */
    MyDacBuffConfigStruct.bufIndexWatermarkIntEnable = true;
    MyDacBuffConfigStruct.bufIndexStartIntEnable = true;
    MyDacBuffConfigStruct.bufIndexUpperIntEnable = true;
    MyDacBuffConfigStruct.dmaEnable = false;
    MyDacBuffConfigStruct.watermarkMode = kDacBuffWatermarkFromUpperAs2Word
    ;
    MyDacBuffConfigStruct.bufWorkMode = kDacBuffWorkAsSwingMode;
    MyDacBuffConfigStruct.bufUpperIndex = buffLen - 1;
    DAC_DRV_EnableBuff(instance, &MyDacBuffConfigStruct, &MyDacStateStructForBufferSwing)
    ;

    /* Fill the buffer with setting data. */
    for (i = 0; i < buffLen; i++)
    {
        DAC_DRV_SetBuffValue(instance, 0U, buffLen, buffPtr);
    }
    /* Register the callback function for DAC buffer event. */
    DAC_DRV_InstallCallback(instance, DAC_ISR_Buffer);

    /* Trigger the buffer to output setting value. */
    for (i = 0U; i < buffLen*4U; i++)
    {
        dacValue = DAC_DRV_SoftTriggerBuff(instance);
        dacValue = dacValue;
        printf("DAC_DRV_SoftTriggerBuff: %d\r\n", dacValue);
        OSA_TimeDelay(200);
    }

    /* Disable the feature of DAC internal buffer. */
    DAC_DRV_DisableBuff(instance);

    /* De-initialize the DAC converter. */
    DAC_DRV_Deinit(instance);
}
```

One-time Scan buffer mode:

```
/* dac_test_buffer_one_time_scan.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_dac_driver.h"
#include "fsl_os_abstraction.h"

static dac_state_t MyDacStateStructForBufferOneTimeScan;
```

```

extern void DAC_ISR_Buffer(void);

void DAC_TEST_BufferOneTimeScanMode(uint32_t instance, uint16_t *buffPtr, uint8_t buffLen)
{
    dac_user_config_t MyDacUserConfigStruct;
    dac_buff_config_t MyDacBuffConfigStruct;
    uint8_t i;
    volatile uint16_t dacValue;

    /* Fill the structure with configuration of software trigger. */
    DAC_DRV_StructInitUserConfigNormal(&MyDacUserConfigStruct);

    /* Initialize the DAC Converter. */
    DAC_DRV_Init(instance, &MyDacUserConfigStruct);

    /* Enable the feature of DAC internal buffer. */
    MyDacBuffConfigStruct	buffIndexWatermarkIntEnable = true;
    MyDacBuffConfigStruct	buffIndexStartIntEnable = true;
    MyDacBuffConfigStruct	buffIndexUpperIntEnable = true;
    MyDacBuffConfigStruct.dmaEnable = false;
    MyDacBuffConfigStruct.watermarkMode = kDacBuffWatermarkFromUpperAs2Word
    ;
    MyDacBuffConfigStruct.buffWorkMode =
        kDacBuffWorkAsOneTimeScanMode;
    MyDacBuffConfigStruct.buffUpperIndex = buffLen - 1;
    DAC_DRV_EnableBuff(instance, &MyDacBuffConfigStruct, &
        MyDacStateStructForBufferOneTimeScan);

    /* Fill the buffer with setting data. */
    DAC_DRV_SetBuffValue(instance, 0U, buffLen, buffPtr);

    /* Register the callback function for DAC buffer event. */
    DAC_DRV_InstallCallback(instance, DAC_ISR_Buffer);

    /* Trigger the buffer to output setting value. */
    for (i = 0U; i < buffLen*4U; i++)
    {
        dacValue = DAC_DRV_SoftTriggerBuff(instance);
        printf("DAC_DRV_SoftTriggerBuff: %d\r\n", dacValue);
        OSA_TimeDelay(200);
    }

    /* Disable the feature of DAC internal buffer. */
    DAC_DRV_DisableBuff(instance);

    /* De-initialize the DAC converter. */
    DAC_DRV_Deinit(instance);
}

```

6.2.1 Data Structure Documentation

6.2.1.1 struct dac_buff_config_t

This structure keeps the configuration for the internal buffer inside the DAC module. The DAC buffer is an advanced feature, which helps improve the performance of an application.

Data Fields

- bool `buffIndexStartIntEnable`
Switcher to enable interrupt when buffer index hits the start (0).

DAC Peripheral Driver

- bool `buffIndexUpperIntEnable`
Switcher to enable interrupt when buffer index hits the upper.
- bool `dmaEnable`
Switcher to enable DMA request by original interrupts.
- `dac_buff_work_mode_t buffWorkMode`
Selection of buffer's work mode, see to "dac_buff_work_mode_t".

6.2.1.1.0.9 Field Documentation

6.2.1.1.0.9.1 `bool dac_buff_config_t::buffIndexStartIntEnable`

6.2.1.1.0.9.2 `bool dac_buff_config_t::buffIndexUpperIntEnable`

6.2.1.1.0.9.3 `bool dac_buff_config_t::dmaEnable`

6.2.1.1.0.9.4 `dac_buff_work_mode_t dac_buff_config_t::buffWorkMode`

6.2.1.2 `struct dac_user_config_t`

This structure keeps the configuration for DAC module basic converter. The DAC converter is the core part of the DAC module. When initialized, the DAC module can act as a simple DAC converter.

Data Fields

- `dac_ref_volt_src_mode_t refVoltSrcMode`
Selection of reference voltage source for DAC module.
- `dac_trigger_mode_t triggerMode`
Selection of hardware mode or software mode.
- bool `lowPowerEnable`
Switcher to enable working in low power mode.

6.2.1.2.0.10 Field Documentation

6.2.1.2.0.10.1 `dac_ref_volt_src_mode_t dac_user_config_t::refVoltSrcMode`

6.2.1.2.0.10.2 `dac_trigger_mode_t dac_user_config_t::triggerMode`

6.2.1.2.0.10.3 `bool dac_user_config_t::lowPowerEnable`

6.2.1.3 `struct dac_state_t`

The contents of this structure are internal to the driver and should not be modified by users.

Data Fields

- `dac_callback_t userCallbackFunc`
Keep the user-defined callback function.

6.2.1.3.0.11 Field Documentation

6.2.1.3.0.11.1 `dac_callback_t dac_state_t::userCallbackFunc`

6.2.2 Enumeration Type Documentation

6.2.2.1 `enum dac_flag_t`

Enumerator

kDacBuffIndexStartFlag Event for the buffer index hit the start (0).

kDacBuffIndexUpperFlag Event for the buffer index hit the upper.

6.2.3 Function Documentation

6.2.3.1 `dac_status_t DAC_DRV_StructInitUserConfigNormal (dac_user_config_t * userConfigPtr)`

This function fills the initial user configuration without the interrupt and buffer features. Calling the initialization function with the filled parameter configures the DAC module to function as a simple converter. The settings are:

```
.refVoltSrcMode = kDacRefVoltSrcOfVref2; // Vdda .triggerMode = kDacTriggerBySoftware; .lowPowerEnable = false;
```

Parameters

<i>userConfigPtr</i>	Pointer to the user configuration structure. See the "dac_user_config_t".
----------------------	---

Returns

Execution status.

6.2.3.2 `dac_status_t DAC_DRV_Init (uint32_t instance, dac_user_config_t * userConfigPtr)`

This function initializes the converter. It configures the DAC converter itself but does not include the advanced features, such as interrupt and internal buffer. This API should be called before any operations in the DAC module. After it is initialized, the DAC module can function as a simple DAC converter.

DAC Peripheral Driver

Parameters

<i>instance</i>	DAC instance ID.
<i>userConfigPtr</i>	Pointer to the initialization structure. See the "dac_user_config_t".

Returns

Execution status.

6.2.3.3 void DAC_DRV_Deinit (uint32_t *instance*)

This function de-initializes the converter. It disables the DAC module and shuts down the clock to reduce the power consumption.

Parameters

<i>instance</i>	DAC instance ID.
-----------------	------------------

6.2.3.4 void DAC_DRV_Output (uint32_t *instance*, uint16_t *value*)

This function drives the converter to output the DAC value. It forces the buffer index to be the first one and load the setting value to this item. Then, the converter outputs the voltage indicated by the indicated value immediately.

Parameters

<i>instance</i>	DAC instance ID.
<i>value</i>	Setting value for DAC.

6.2.3.5 dac_status_t DAC_DRV_EnableBuff (uint32_t *instance*, dac_buff_config_t * *buffConfigPtr*, dac_state_t * *userStatePtr*)

This function configures the feature of the internal buffer for the DAC module. By default, the buffer feature is disabled. Calling this API enables the buffer and configures it.

Parameters

<i>instance</i>	DAC instance ID.
<i>buffConfigPtr</i>	Pointer to the configuration structure. See the "dac_buff_config_t".
<i>userStatePtr</i>	Pointer to the structure for keeping internal state. See the "dac_state_t".

Returns

Execution status.

6.2.3.6 void DAC_DRV_DisableBuff (uint32_t *instance*)

This function disables the internal buffer feature. Calling this API disables the internal buffer feature and resets the DAC module as a simple DAC converter.

Parameters

<i>instance</i>	DAC instance ID.
-----------------	------------------

6.2.3.7 dac_status_t DAC_DRV_SetBuffValue (uint32_t *instance*, uint8_t *start*, uint8_t *offset*, uint16_t *arr*[])

This function sets values into the DAC internal buffer. Note that the buffer size is defined by the "FSL_FEATURE_DAC_BUFFER_SIZE" macro and the available value is 12 bit.

Parameters

<i>instance</i>	DAC instance ID.
<i>start</i>	Start index of setting values.
<i>offset</i>	Length of setting values' array.
<i>arr</i>	Setting values' array.

Returns

Execution status.

6.2.3.8 uint16_t DAC_DRV_SoftTriggerBuff (uint32_t *instance*)

This function triggers the buffer by software and returns the current value. After it is triggered, the buffer index updates according to work mode. Then, the value kept inside the pointed item is immediately output.

DAC Peripheral Driver

Parameters

<i>instance</i>	DAC instance ID.
-----------------	------------------

Returns

Current output value.

6.2.3.9 **uint8_t DAC_DRV_GetBufferIndex (uint32_t *instance*)**

This function gets the DAC buffer current index.

Parameters

<i>instance</i>	DAC instance ID.
-----------------	------------------

Returns

Current index of DAC buffer.

6.2.3.10 **void DAC_DRV_ClearFlag (uint32_t *instance*, dac_flag_t *flag*)**

This function clears the flag for an indicated event causing an interrupt.

Parameters

<i>instance</i>	DAC instance ID.
<i>flag</i>	Indicated flag, see to "dac_flag_t".

6.2.3.11 **bool DAC_DRV_GetFlag (uint32_t *instance*, dac_flag_t *flag*)**

This function gets the flag for an indicated event causing an interrupt. If the event occurs, the return value is asserted.

Parameters

<i>instance</i>	DAC instance ID.
<i>flag</i>	Indicated flag, see to "dac_flag_t".

Returns

Assertion of indicated event.

6.2.3.12 **dac_status_t DAC_DRV_InstallCallback (uint32_t *instance*, dac_callback_t *userCallback*)**

This function installs the user-defined callback. When the DAC interrupt request is served, the callback is executed inside the ISR.

DAC Peripheral Driver

Parameters

<i>instance</i>	DAC instance ID.
<i>userCallback</i>	User-defined callback function.

Returns

Execution status.

6.2.3.13 void DAC_DRV_IRQHandler (uint32_t *instance*)

This function is the driver-defined ISR in DAC module. It includes the process for interrupt mode defined by the driver. Currently, it is called inside the system-defined ISR.

Parameters

<i>instance</i>	DAC instance ID.
-----------------	------------------

Chapter 7

Direct Memory Access (DMA)

The Kinetis SDK provides both HAL and Peripheral drivers for the Direct Memory Access block of Kinetis devices.

Modules

- [DMA Driver](#)

This part describes the programming interface of the DMA Peripheral driver.

- [DMA HAL driver](#)

This part describes the programming interface of the DMA HAL driver.

- [DMA request](#)

This part provides the DMA request resource.

- [DMAMUX HAL driver](#)

This part describes the programming interface of the DMAMUX HAL module.

DMA HAL driver

7.1 DMA HAL driver

This chapter describes the programming interface of the DMA HAL driver.

Data Structures

- struct `dma_channel_link_config_t`
Data structure for data structure configuration. [More...](#)
- union `dma_error_status_t`
Data structure to get status of the DMA channel status. [More...](#)

Enumerations

- enum `dma_status_t` { ,
 `kStatus_DMA_InvalidArgument` = 1U,
 `kStatus_DMA_Fail` = 2U }
DMA status.
- enum `dma_transfer_size_t` {
 `kDmaTransferSize32bits` = 0x0U,
 `kDmaTransferSize8bits` = 0x1U,
 `kDmaTransferSize16bits` = 0x2U }
DMA transfer size type.
- enum `dma_modulo_t`
Configuration type for the DMA modulo.
- enum `dma_channel_link_type_t` {
 `kDmaChannelLinkDisable` = 0x0U,
 `kDmaChannelLinkChan1AndChan2` = 0x1U,
 `kDmaChannelLinkChan1` = 0x2U,
 `kDmaChannelLinkChan1AfterBCR0` = 0x3 }
DMA channel link type.
- enum `dma_transfer_type_t` {
 `kDmaPeripheralToMemory`,
 `kDmaMemoryToPeripheral`,
 `kDmaMemoryToMemory`,
 `kDmaPeripheralToPeripheral` }
Type for DMA transfer.

DMA HAL channel configuration

- void `DMA_HAL_Init` (uint32_t baseAddr, uint32_t channel)
Sets all registers of the channel to 0.
- void `DMA_HAL_ConfigTransfer` (uint32_t baseAddr, uint32_t channel, `dma_transfer_size_t` size, `dma_transfer_type_t` type, uint32_t sourceAddr, uint32_t destAddr, uint32_t length)
Sets all registers of the channel to 0.
- static void `DMA_HAL_SetSourceAddr` (uint32_t baseAddr, uint32_t channel, uint32_t address)

- static void **DMA_HAL_SetDestAddr** (uint32_t baseAddr, uint32_t channel, uint32_t address)
Configures the source address.
- static void **DMA_HAL_SetTransferCount** (uint32_t baseAddr, uint32_t channel, uint32_t count)
Configures the bytes to be transferred.
- static uint32_t **DMA_HAL_GetUnfinishedByte** (uint32_t baseAddr, uint32_t channel)
Gets the left bytes not to be transferred.
- static void **DMA_HAL_SetIntCmd** (uint32_t baseAddr, uint8_t channel, bool enable)
Enables the interrupt for the DMA channel after the work is done.
- static void **DMA_HAL_SetCycleStealCmd** (uint32_t baseAddr, uint8_t channel, bool enable)
Configures the DMA transfer mode to cycle steal or continuous modes.
- static void **DMA_HAL_SetAutoAlignCmd** (uint32_t baseAddr, uint8_t channel, bool enable)
Configures the auto-align feature.
- static void **DMA_HAL_SetAsyncDmaRequestCmd** (uint32_t baseAddr, uint8_t channel, bool enable)
Configures the a-sync DMA request feature.
- static void **DMA_HAL_SetSourceIncrementCmd** (uint32_t baseAddr, uint32_t channel, bool enable)
Enables/Disables the source increment.
- static void **DMA_HAL_SetDestIncrementCmd** (uint32_t baseAddr, uint32_t channel, bool enable)
Enables/Disables destination increment.
- static void **DMA_HAL_SetSourceTransferSize** (uint32_t baseAddr, uint32_t channel, **dma_transfer_size_t** transfersize)
Configures the source transfer size.
- static void **DMA_HAL_SetDestTransferSize** (uint32_t baseAddr, uint32_t channel, **dma_transfer_size_t** transfersize)
Configures the destination transfer size.
- static void **DMA_HAL_SetTriggerStartCmd** (uint32_t baseAddr, uint32_t channel, bool enable)
Triggers the start.
- static void **DMA_HAL_SetSourceModulo** (uint32_t baseAddr, uint32_t channel, **dma_modulo_t** modulo)
Configures the modulo for the source address.
- static void **DMA_HAL_SetDestModulo** (uint32_t baseAddr, uint32_t channel, **dma_modulo_t** modulo)
Configures the modulo for the destination address.
- static void **DMA_HAL_SetDmaRequestCmd** (uint32_t baseAddr, uint32_t channel, bool enable)
Enables/Disables the DMA request.
- static void **DMA_HAL_SetDisableRequestAfterDoneCmd** (uint32_t baseAddr, uint32_t channel, bool enable)
Configures the DMA request state after the work is done.
- void **DMA_HAL_SetChanLink** (uint32_t baseAddr, uint8_t channel, **dma_channel_link_config_t** *mode)
Configures the channel link feature.
- static void **DMA_HAL_ClearStatus** (uint32_t baseAddr, uint8_t channel)
Clears the status of the DMA channel.
- static **dma_error_status_t** **DMA_HAL_GetStatus** (uint32_t baseAddr, uint8_t channel)
Gets the DMA controller channel status.

DMA HAL driver

7.1.0.14 DMA HAL Driver

Overview

The DMA HAL driver masks the hardware and provide user a comprehensible way to use the DMA.

7.1.1 Data Structure Documentation

7.1.1.1 `struct dma_channel_link_config_t`

Data Fields

- `dma_channel_link_type_t linkType`
Channel link type.
- `uint32_t channel1`
Channel 1 configuration.
- `uint32_t channel2`
Channel 2 configuration.

7.1.1.2 `union dma_error_status_t`

7.1.1.2.0.12 Field Documentation

7.1.1.2.0.12.1 `uint32_t dma_error_status_t::dmaTransDone`

7.1.1.2.0.12.2 `uint32_t dma_error_status_t::dmaBusy`

7.1.1.2.0.12.3 `uint32_t dma_error_status_t::dmaPendingRequest`

7.1.2 Enumeration Type Documentation

7.1.2.1 `enum dma_status_t`

Enumerator

kStatus_DMA_InvalidArgument Parameter is not available for the current configuration.

kStatus_DMA_Fail Function operation failed.

7.1.2.2 `enum dma_transfer_size_t`

Enumerator

kDmaTransferSize32bits 32 bits are transferred for every read/write

kDmaTransferSize8bits 8 bits are transferred for every read/write

kDmaTransferSize16bits 16b its are transferred for every read/write

7.1.2.3 enum dma_channel_link_type_t

Enumerator

kDmaChannelLinkDisable No channel link.

kDmaChannelLinkChan1AndChan2 Perform a link to channel 1 after each cycle-steal transfer followed by a link and to channel 2 after the BCR decrements to zeros.

kDmaChannelLinkChan1 Perform a link to channel 1 after each cycle-steal transfer.

kDmaChannelLinkChan1AfterBCR0 Perform a link to channel1 after the BCR decrements to zero.

7.1.2.4 enum dma_transfer_type_t

Enumerator

kDmaPeripheralToMemory Transfer from the peripheral to memory.

kDmaMemoryToPeripheral Transfer from the memory to peripheral.

kDmaMemoryToMemory Transfer from the memory to memory.

kDmaPeripheralToPeripheral Transfer from the peripheral to peripheral.

7.1.3 Function Documentation

7.1.3.1 void DMA_HAL_Init (uint32_t *baseAddr*, uint32_t *channel*)

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.

7.1.3.2 void DMA_HAL_ConfigTransfer (uint32_t *baseAddr*, uint32_t *channel*, dma_transfer_size_t *size*, dma_transfer_type_t *type*, uint32_t *sourceAddr*, uint32_t *destAddr*, uint32_t *length*)

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.

DMA HAL driver

<i>size</i>	Size to be transferred on each DMA write/read. Source/Dest share the same write/read size.
<i>type</i>	Transfer type.
<i>sourceAddr</i>	Source address.
<i>destAddr</i>	Destination address.
<i>length</i>	Bytes to be transferred.

7.1.3.3 static void DMA_HAL_SetSourceAddr (*uint32_t baseAddr, uint32_t channel, uint32_t address*) [inline], [static]

Each SAR contains the byte address used by the DMA to read data. The SARn is typically aligned on a 0-modulo-size boundary—that is on the natural alignment of the source data. Bits 31-20 of this register must be written with one of the only four allowed values. Each of these allowed values corresponds to a valid region of the devices' memory map. The allowed values are: 0x000x_xxxx 0x1FFx_xxxx 0x200x_xxxx 0x400x_xxxx After they are written with one of the allowed values, bits 31-20 read back as the written value. After they are written with any other value, bits 31-20 read back as an indeterminate value.

This function enables the request for a specified channel.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>address</i>	memory address pointing to the source address.

7.1.3.4 static void DMA_HAL_SetDestAddr (*uint32_t baseAddr, uint32_t channel, uint32_t address*) [inline], [static]

Each DAR contains the byte address used by the DMA to read data. The DARn is typically aligned on a 0-modulo-size boundary—that is on the natural alignment of the source data. Bits 31-20 of this register must be written with one of the only four allowed values. Each of these allowed values corresponds to a valid region of the devices' memory map. The allowed values are: 0x000x_xxxx 0x1FFx_xxxx 0x200x_xxxx 0x400x_xxxx After they are written with one of the allowed values, bits 31-20 read back as the written value. After they are written with any other value, bits 31-20 read back as an indeterminate value.

This function enables the request for specified channel.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.

7.1.3.5 static void DMA_HAL_SetTransferCount(*uint32_t baseAddr, uint32_t channel, uint32_t count*) [inline], [static]

Transfer bytes must be written with a value equal to or less than 0F_FFFFh. After being written with a value in this range, bits 23-20 of the BCR read back as 1110b. A write to the BCR with a value greater than 0F_FFFFh causes a configuration error when the channel starts to execute. After they are written with a value in this range, bits 23-20 of BCR read back as 1111b.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>count</i>	bytes to be transferred.

7.1.3.6 static uint32_t DMA_HAL_GetUnfinishedByte(*uint32_t baseAddr, uint32_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.

Returns

unfinished bytes.

7.1.3.7 static void DMA_HAL_SetIntCmd(*uint32_t baseAddr, uint8_t channel, bool enable*) [inline], [static]

This function enables the request for specified channel.

DMA HAL driver

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.

7.1.3.8 static void DMA_HAL_SetCycleStealCmd (*uint32_t baseAddr, uint8_t channel, bool enable*) [inline], [static]

If continuous mode is enabled, DMA continuously makes write/read transfers until BCR decrement to 0. If continuous mode is disabled, DMA write/read is only triggered on every request. s

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>isContinue</i>	0 means cycle-steal mode, 1 means continuous mode.

7.1.3.9 static void DMA_HAL_SetAutoAlignCmd (*uint32_t baseAddr, uint8_t channel, bool enable*) [inline], [static]

If auto-align is enabled, the appropriate address register increments, regardless of whether it is a source increment or a destination increment.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>isEnable</i>	0 means disable auto-align. 1 means enable auto-align.

7.1.3.10 static void DMA_HAL_SetAsyncDmaRequestCmd (*uint32_t baseAddr, uint8_t channel, bool enable*) [inline], [static]

Enables/Disables the a-synchronization mode in a STOP mode for each DMA channel.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>isEnable</i>	0 means disable DMA request a-sync. 1 means enable DMA request -.

7.1.3.11 static void DMA_HAL_SetSourceIncrementCmd (uint32_t *baseAddr*, uint32_t *channel*, bool *enable*) [inline], [static]

Controls whether the source address increments after each successful transfer. If enabled, the SAR increments by 1,2,4 as determined by the transfer size.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>isEnabled</i>	Enabled/Disable increment.

7.1.3.12 static void DMA_HAL_SetDestIncrementCmd (uint32_t *baseAddr*, uint32_t *channel*, bool *enable*) [inline], [static]

Controls whether the destination address increments after each successful transfer. If enabled, the DAR increments by 1,2,4 as determined by the transfer size.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>isEnabled</i>	Enabled/Disable increment.

7.1.3.13 static void DMA_HAL_SetSourceTransferSize (uint32_t *baseAddr*, uint32_t *channel*, dma_transfer_size_t *transfersize*) [inline], [static]

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>transfersize</i>	enum type for transfer size.

DMA HAL driver

7.1.3.14 **static void DMA_HAL_SetDestTransferSize (uint32_t *baseAddr*, uint32_t *channel*, dma_transfer_size_t *transfersize*) [inline], [static]**

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>transfersize</i>	enum type for transfer size.

7.1.3.15 static void DMA_HAL_SetTriggerStartCmd (*uint32_t baseAddr, uint32_t channel, bool enable*) [inline], [static]

When the DMA begins the transfer, the START bit is cleared automatically after one module clock and always reads as logic 0.

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.

7.1.3.16 static void DMA_HAL_SetSourceModulo (*uint32_t baseAddr, uint32_t channel, dma_modulo_t modulo*) [inline], [static]

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>modulo</i>	enum data type for source modulo.

7.1.3.17 static void DMA_HAL_SetDestModulo (*uint32_t baseAddr, uint32_t channel, dma_modulo_t modulo*) [inline], [static]

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>modulo</i>	enum data type for dest modulo.

7.1.3.18 static void DMA_HAL_SetDmaRequestCmd (*uint32_t baseAddr, uint32_t channel, bool enable*) [inline], [static]

DMA HAL driver

Parameters

<i>baseAddr</i>	DMA baseAddr.
<i>channel</i>	DMA channel.
<i>isEnabled</i>	

7.1.3.19 static void DMA_HAL_SetDisableRequestAfterDoneCmd (*uint32_t baseAddr, uint32_t channel, bool enable*) [inline], [static]

Disables/Enables the DMA request after a DMA DONE is generated. If it works in the loop mode, this bit should not be set.

Parameters

<i>channel</i>	DMA channel.
<i>isDisabled</i>	0 means DMA request would not be disabled after work done. 1 means disable.

7.1.3.20 void DMA_HAL_SetChanLink (*uint32_t baseAddr, uint8_t channel, dma_channel_link_config_t * mode*)

Parameters

<i>channel</i>	DMA channel.
<i>mode</i>	Mode of channel link in DMA.

7.1.3.21 static void DMA_HAL_ClearStatus (*uint32_t baseAddr, uint8_t channel*) [inline], [static]

This function clears the status for a specified DMA channel. The error status and done status are cleared.

Parameters

<i>channel</i>	DMA channel.
----------------	--------------

7.1.3.22 static dma_error_status_t DMA_HAL_GetStatus (*uint32_t baseAddr, uint8_t channel*) [inline], [static]

Gets the status of the DMA channel. The user can get the error status, as to whether the descriptor is finished or there are bytes left.

Parameters

<i>channel</i>	DMA channel.
<i>mode</i>	Mode of channel link in DMA.

Returns

Status of the DMA channel.

Function Documentation

7.2 DMAMUX HAL driver

This chapter describes the programming interface of the DMAMUX HAL module.

Enumerations

- enum `dmamux_dma_request_source` { `kDmamuxDmaRequestSource` = 64U }

A constant for the length of the DMA hardware source.

DMAMUX HAL function

- void `DMAMUX_HAL_Init` (uint32_t `baseAddr`)
Initializes the DMAMUX module to the reset state.
- static void `DMAMUX_HAL_SetChannelCmd` (uint32_t `baseAddr`, uint32_t `channel`, bool `enable`)
Enables/Disables the DMAMUX channel.
- static void `DMAMUX_HAL_SetPeriodTriggerCmd` (uint32_t `baseAddr`, uint32_t `channel`, bool `enable`)
Enables/Disables the period trigger.
- static void `DMAMUX_HAL_SetTriggerSource` (uint32_t `baseAddr`, uint32_t `channel`, uint8_t `source`)
Configures the DMA request for the DMAMUX channel.

7.3 Enumeration Type Documentation

7.3.1 enum dmamux_dma_request_source

This structure is used inside the DMA driver.

Enumerator

`kDmamuxDmaRequestSource` Maximum number of the DMA requests allowed for the DMA mux.

7.4 Function Documentation

7.4.1 void `DMAMUX_HAL_Init` (`uint32_t baseAddr`)

Initializes the DMAMUX module to the reset state.

Parameters

<code>baseAddr</code>	Register base address for DMAMUX module.
-----------------------	--

7.4.2 static void `DMAMUX_HAL_SetChannelCmd` (`uint32_t baseAddr`, `uint32_t channel`, `bool enable`) [inline], [static]

Enables the hardware request. If enabled, the hardware request is sent to the corresponding DMA channel.

Parameters

<i>baseAddr</i>	Register base address for DMAMUX module.
<i>channel</i>	DMAMUX channel number.
<i>enable</i>	Enables (true) or Disables (false) DMAMUX channel.

7.4.3 static void DMAMUX_HAL_SetPeriodTriggerCmd (*uint32_t baseAddr, uint32_t channel, bool enable*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for DMAMUX module.
<i>channel</i>	DMAMUX channel number.
<i>enable</i>	Enables (true) or Disables (false) period trigger.

7.4.4 static void DMAMUX_HAL_SetTriggerSource (*uint32_t baseAddr, uint32_t channel, uint8_t source*) [inline], [static]

Sets the trigger source for the DMA channel. The trigger source is in the file fsl_dma_request.h.

Parameters

<i>baseAddr</i>	Register base address for DMAMUX module.
<i>channel</i>	DMAMUX channel number.
<i>source</i>	DMA request source.

DMA Driver

7.5 DMA Driver

This chapter describes the programming interface of the DMA Peripheral driver.

Data Structures

- struct `dma_channel_t`
Data structure for the DMA channel management. [More...](#)
- struct `dma_state_t`
Data structure for the DMA controller management. [More...](#)

TypeDefs

- `typedef void(* dma_callback_t)(void *parameter, dma_channel_status_t status)`
A definition for the DMA channel callback function.

Enumerations

- enum `dma_channel_status_t` {
 `kDmaIdle`,
 `kDmaNormal`,
 `kDmaError` }
Channel status for the DMA channel.
- enum `dma_channel_type_t` {
 `kDmaInvalidChannel` = 0xFFU,
 `kDmaAnyChannel` = 0xFEU }
Type for the DMA channel, which is used for the DMA channel allocation.

DMA Driver

- `dma_status_t DMA_DRV_Init (dma_state_t *state)`
Initializes the DMA.
- `dma_status_t DMA_DRV_Deinit (void)`
De-initializes the DMA.
- `dma_status_t DMA_DRV_RegisterCallback (dma_channel_t *chn, dma_callback_t callback, void *para)`
Registers the callback function and a parameter.
- `uint32_t DMA_DRV_GetUnfinishedBytes (dma_channel_t *chn)`
Gets the number of unfinished bytes.
- `dma_status_t DMA_DRV_ClaimChannel (uint32_t channel, dma_request_source_t source, dma_channel_t *chn)`
Gets the status of the EDMA channel descriptor chain.
- `uint32_t DMA_DRV_RequestChannel (uint32_t channel, dma_request_source_t source, dma_channel_t *chn)`
Requests a DMA channel.

- **dma_status_t DMA_DRV_FreeChannel (dma_channel_t *chn)**
Frees DMA channel hardware and software resource.
- **dma_status_t DMA_DRV_StartChannel (dma_channel_t *chn)**
Starts a DMA channel.
- **dma_status_t DMA_DRV_StopChannel (dma_channel_t *chn)**
Stops a DMA channel.
- **dma_status_t DMA_DRV_ConfigTransfer (dma_channel_t *chn, dma_transfer_type_t type, uint32_t size, uint32_t sourceAddr, uint32_t destAddr, uint32_t length)**
Configures a transfer for the DMA.
- **dma_status_t DMA_DRV_ConfigChanLink (dma_channel_t *chn, dma_channel_link_config_t *link_config)**
Configures the channel link feature.
- **void DMA_DRV_IRQHandler (uint32_t channel)**
DMA IRQ handler for both an interrupt and an error.

7.5.0.1 DMA Peripheral Driver

Overview

The DMA driver requests, configures, and uses the DMA hardware. It supports module initializations and DMA channel configurations.

Initialization

To initialize the DMA module, call the `dma_init()` function. Configuration data structure does not need to be passed. This function enables the DMA module and clock automatically.

Channel concept

DMA module consists of many channels. The DMA Peripheral driver is designed based on the channel concept. All tasks should start by requesting a DMA channel and end by freeing a DMA channel. By getting a channel allocated, the user can configure and run operations on the DMA module. If a channel is not allocated, a system error may occur.

DMA request concept

DMA request triggers a DMA transfer. The DMA request table is available in the chip configuration chapters in a Reference Manual.

DMA Driver

Memory allocation

DMA peripheral driver does not allocate memory dynamically. The user must provide the allocated memory pointer for the driver and ensure that the memory is valid. Otherwise, a system error occurs. The user needs to provide the memory for the [dma_channel_t]. The driver must store the status data for every channel and the [dma_channel_t](#) is designed for this purpose.

Call diagram

To use the DMA driver, follow these steps:

1. Initialize the DMA module: `dma_init()`.
2. Request a DMA channel: `dma_request_channel()`.
3. Configure the TCD: `dma_config_transfer()`.
4. Register callback function: `dma_register_callback()`.
5. Start the DMA channel: `dma_start_channel()`.
6. [OPTION] Stop the DMA channel: `dma_stop_channel()`.
7. Free the DMA channel: `dma_free_channel()`.

This is an example code to initialize and configure a memory-to-memory transfer:

```
uint32_t j, temp;
dma_channel_t chan_handler;
uint8_t *srcAddr, *destAddr;
fsl_rtos_status syncStatus;

srcAddr = malloc(kDmaTestBufferSize);
destAddr = malloc(kDmaTestBufferSize);
if (((uint32_t)srcAddr == 0x0U) & ((uint32_t)destAddr == 0x0U))
{
    printf("Failed to allocate memory for test! \r\n");
    goto error;
}

/* Init the memory buffer. */
for (j = 0; j < kDmaTestBufferSize; j++)
{
    srcAddr[j] = j;
    destAddr[j] = 0;
}

temp = dma_request_channel(channel, kDmaRequestMux0AlwaysOn62, &chan_handler);
if (temp != channel)
{
    printf("Failed to request channel %d !\r\n", channel);
    goto error;
}

dma_config_transfer(&chan_handler, kDmaMemoryToMemory,
                    0x1U,
                    (uint32_t)srcAddr, (uint32_t)destAddr,
                    kDmaTestBufferSize);

dma_register_callback(&chan_handler, test_callback, &chan_handler);

dma_start_channel(&chan_handler);
//Wait until channel complete...
```

```
dma_stop_channel(&chan_handler);
dma_free_channel(&chan_handler);
```

7.5.1 Data Structure Documentation

7.5.1.1 struct dma_channel_t

Data Fields

- uint8_t **channel**
Channel number.
- uint8_t **dmamuxModule**
Dmamux module index.
- uint8_t **dmamuxChannel**
Dmamux module channel.
- **dma_callback_t callback**
Callback function for this channel.
- void * **parameter**
Parameter for the callback function.
- volatile **dma_channel_status_t status**
Channel status.

7.5.1.2 struct dma_state_t

7.5.2 Typedef Documentation

7.5.2.1 **typedef void(* dma_callback_t)(void *parameter, dma_channel_status_t status)**

A prototype for the callback function registered into the DMA driver.

7.5.3 Enumeration Type Documentation

7.5.3.1 enum dma_channel_status_t

A structure describing the status of the DMA channel. The user can get the status from the channel callback function.

Enumerator

kDmaIdle DMA channel is idle.

kDmaNormal DMA channel is occupied.

kDmaError Error occurs in the DMA channel.

7.5.3.2 enum dma_channel_type_t

Enumerator

kDmaInvalidChannel Macros indicating the failure of the channel request.

kDmaAnyChannel Macros used when requesting a channel. kEdmaAnyChannel means a request of dynamic channel allocation.

7.5.4 Function Documentation

7.5.4.1 dma_status_t DMA_DRV_RegisterCallback (**dma_channel_t * chn,** **dma_callback_t callback, void * para**)

The user registers the callback function and a parameter for a specified DMA channel. When the channel interrupt or a channel error happens, the callback and the parameter are called. The user parameter is also provided to give a channel status.

Parameters

<i>chn</i>	A handler for the DMA channel
<i>callback</i>	Callback function
<i>para</i>	A parameter for callback functions

7.5.4.2 uint32_t DMA_DRV_GetUnfinishedBytes (**dma_channel_t * chn**)

Gets the left bytes to be transferred.

Parameters

<i>chn</i>	A handler for the DMA channel
------------	-------------------------------

7.5.4.3 dma_status_t DMA_DRV_ClaimChannel (**uint32_t channel,** **dma_request_source_t source, dma_channel_t * chn**)

Gets the left bytes to be transferred.

Parameters

<i>chn</i>	A handler for the DMA channel
------------	-------------------------------

7.5.4.4 **uint32_t DMA_DRV_RequestChannel (uint32_t *channel*, dma_request_source_t *source*, dma_channel_t * *chn*)**

This function provides two ways to allocate a DMA channel. The first way is a static allocation. The second way is a dynamic allocation. To allocate a channel dynamically, the user needs to set the channel parameter with the value of kDmaAnyChannel. The driver searches into all available free channels and assigns the first channel to the user. To allocate the channel statically, the user needs to set the channel parameter with the value of a specified channel. If the channel is available, the driver assigns the channel to the user. Notes: The user must provide a handler memory for the DMA channel. The driver initializes the handler and configures the handler memory.

Parameters

<i>channel</i>	A DMA channel number. If a channel is assigned with a valid channel number, the DMA driver tries to assign a specified channel to the user. If a channel is assigned with kDmaAnyChannel, the DMA driver searches all available channels and assigns the first channel to the user.
<i>source</i>	A DMA hardware request.
<i>chan</i>	Memory pointing to DMA channel. The user must ensure that the handler memory is valid and that it will not be released or changed by any other codes before the channel <code>dma_free_channel()</code> operation.

Returns

If the channel allocation is successful, the return value indicates the requested channel. If not, the driver returns a kDmaInvalidChannel value to indicate that the request operation has failed.

7.5.4.5 **dma_status_t DMA_DRV_FreeChannel (dma_channel_t * *chn*)**

This function frees the relevant software and hardware resources. Both the request and the free operations need to be called in a pair.

Parameters

DMA Driver

<i>chn</i>	Memory pointing to DMA channel.
------------	---------------------------------

7.5.4.6 **dma_status_t DMA_DRV_StartChannel (dma_channel_t * *chn*)**

Starts a DMA channel. The driver starts a DMA channel by enabling the DMA request. A software start bit is not used in the DMA Peripheral driver.

Parameters

<i>chn</i>	Memory pointing to the DMA channel.
------------	-------------------------------------

7.5.4.7 **dma_status_t DMA_DRV_StopChannel (dma_channel_t * *chn*)**

Parameters

<i>chn</i>	Memory pointing to the DMA channel.
------------	-------------------------------------

7.5.4.8 **dma_status_t DMA_DRV_ConfigTransfer (dma_channel_t * *chn*, dma_transfer_type_t *type*, uint32_t *size*, uint32_t *sourceAddr*, uint32_t *destAddr*, uint32_t *length*)**

Configures a transfer for the DMA.

Parameters

<i>chn</i>	Memory pointing to the DMA channel.
<i>type</i>	Transfer type.
<i>size</i>	Size to be transferred on each DMA write/read. Source/Dest share the same write/read size.
<i>sourceAddr</i>	Source address.
<i>destAddr</i>	Destination address.
<i>length</i>	Bytes to be transferred.

7.5.4.9 **dma_status_t DMA_DRV_ConfigChanLink (dma_channel_t * *chn*, dma_channel_link_config_t * *link_config*)**

Parameters

<i>chn</i>	Memory pointing to the DMA channel.
<i>mode</i>	Mode of channel link in DMA.

7.5.4.10 void DMA_DRV_IRQHandler (uint32_t *channel*)

Parameters

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

DMA request

7.6 DMA request

This chapter provides the DMA request resource.

Chapter 8

Serial Peripheral Interface (DSPI)

The Kinetis SDK provides both HAL and Peripheral drivers for the Serial Peripheral Interface (DSPI) block of Kinetis devices.

Modules

- [DSPI HAL driver](#)

This part describes the programming interface of the DSPI HAL driver.

- [DSPI Master Driver](#)

This part describes the programming interface of the DSPI master mode Peripheral driver.

- [DSPI Shared IRQ Driver](#)

This part describes the programming interface of the DSPI shared IRQ driver for master and slave Peripheral drivers.

- [DSPI Slave Driver](#)

This part describes the programming interface of the DSPI slave mode Peripheral driver.

8.1 DSPI HAL driver

This chapter describes the programming interface of the DSPI HAL driver.

Data Structures

- struct `dspi_data_format_config_t`
DSPI data format settings configuration structure. [More...](#)
- struct `dspi_slave_config_t`
DSPI hardware configuration settings for slave mode. [More...](#)
- struct `dspi_baud_rate_divisors_t`
DSPI baud rate divisors settings configuration structure. [More...](#)
- struct `dspi_command_config_t`
DSPI command and data configuration structure. [More...](#)

Enumerations

- enum `dspi_status_t` { ,
 `kStatus_DSPI_SlaveTxUnderrun`,
 `kStatus_DSPI_SlaveRxOverrun`,
 `kStatus_DSPI_Timeout`,
 `kStatus_DSPI_Busy`,
 `kStatus_DSPI_NoTransferInProgress`,
 `kStatus_DSPI_InvalidBitCount`,
 `kStatus_DSPI_InvalidInstanceNumber`,
 `kStatus_DSPI_OutOfRange` }
Error codes for the DSPI driver.
- enum `dspi_master_slave_mode_t` {
 `kDspiMaster` = 1,
 `kDspiSlave` = 0 }
DSPI master or slave configuration.
- enum `dspi_clock_polarity_t` {
 `kDspiClockPolarity_ActiveHigh` = 0,
 `kDspiClockPolarity_ActiveLow` = 1 }
DSPI clock polarity configuration for a given CTAR.
- enum `dspi_clock_phase_t` {
 `kDspiClockPhase_FirstEdge` = 0,
 `kDspiClockPhase_SecondEdge` = 1 }
DSPI clock phase configuration for a given CTAR.
- enum `dspi_shift_direction_t` {
 `kDspiMsbFirst` = 0,
 `kDspiLsbFirst` = 1 }
DSPI data shifter direction options for a given CTAR.
- enum `dspi_ctar_selection_t` {
 `kDspiCtar0` = 0,
 `kDspiCtar1` = 1 }

DSPI Clock and Transfer Attributes Register (CTAR) selection.

- enum `dspi_pcs_polarity_config_t` {

 `kDspiPcs_ActiveHigh` = 0,

 `kDspiPcs_ActiveLow` = 1 }

DSPI Peripheral Chip Select (PCS) Polarity configuration.

- enum `dspi_which_pcs_config_t` {

 `kDspiPcs0` = 1 << 0,

 `kDspiPcs1` = 1 << 1,

 `kDspiPcs2` = 1 << 2,

 `kDspiPcs3` = 1 << 3,

 `kDspiPcs4` = 1 << 4,

 `kDspiPcs5` = 1 << 5 }

DSPI Peripheral Chip Select (PCS) configuration (which PCS to configure)

- enum `dspi_master_sample_point_t` {

 `kDspiSckToSin_0Clock` = 0,

 `kDspiSckToSin_1Clock` = 1,

 `kDspiSckToSin_2Clock` = 2 }

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

- enum `dspi_fifo_t` {

 `kDspiTxFifo` = 0,

 `kDspiRxFifo` = 1 }

DSPI FIFO selects.

- enum `dspi_dma_or_int_mode_t` {

 `kDspiGenerateIntReq` = 0,

 `kDspiGenerateDmaReq` = 1 }

DSPI Tx FIFO Fill and Rx FIFO Drain DMA or Interrupt configuration.

- enum `dspi_status_and_interrupt_request_t` {

 `kDspiTxComplete` = BP_SPI_RSER_TCF_RE,

 `kDspiTxAndRxStatus` = BP_SPI_SR_TXRXS,

 `kDspiEndOfQueue` = BP_SPI_RSER_EOQF_RE,

 `kDspiTxFifoUnderflow` = BP_SPI_RSER_TFUF_RE,

 `kDspiTxFifoFillRequest` = BP_SPI_RSER_TFFF_RE,

 `kDspiRxFifoOverflow` = BP_SPI_RSER_RFOF_RE,

 `kDspiRxFifoDrainRequest` = BP_SPI_RSER_RFDF_RE }

DSPI status flags and interrupt request enable.

- enum `dspi_fifo_counter_pointer_t` {

 `kDspiRxFifoPointer` = BP_SPI_SR_POPNXTPTR,

 `kDspiRxFifoCounter` = BP_SPI_SR_RXCTR,

 `kDspiTxFifoPointer` = BP_SPI_SR_TXNXTPTR,

 `kDspiTxFifoCounter` = BP_SPI_SR_TXCTR }

DSPI FIFO counter or pointer defines based on bit positions.

- enum `dspi_delay_type_t` {

 `kDspiPcsToSck` = 1,

 `kDspiLastSckToPcs` = 2,

 `kDspiAfterTransfer` = 3 }

DSPI delay type selection.

Configuration

- void [DSPI_HAL_Init](#) (uint32_t baseAddr)
Restores the DSPI to reset the configuration.
- static void [DSPI_HAL_Enable](#) (uint32_t baseAddr)
Enables the DSPI peripheral and sets the MCR MDIS to 0.
- static void [DSPI_HAL_Disable](#) (uint32_t baseAddr)
Disables the DSPI peripheral, sets MCR MDIS to 1.
- uint32_t [DSPI_HAL_SetBaudRate](#) (uint32_t baseAddr, [dspi_ctar_selection_t](#) whichCtar, uint32_t bitsPerSec, uint32_t sourceClockInHz)
Sets the DSPI baud rate in bits per second.
- void [DSPI_HAL_SetBaudDivisors](#) (uint32_t baseAddr, [dspi_ctar_selection_t](#) whichCtar, const [dspi_baud_rate_divisors_t](#) *divisors)
Configures the baud rate divisors manually.
- static void [DSPI_HAL_SetMasterSlaveMode](#) (uint32_t baseAddr, [dspi_master_slave_mode_t](#) mode)
Configures the DSPI for master or slave.
- static bool [DSPI_HAL_IsMaster](#) (uint32_t baseAddr)
Returns whether the DSPI module is in master mode.
- static void [DSPI_HAL_SetContinuousSckCmd](#) (uint32_t baseAddr, bool enable)
Configures the DSPI for the continuous SCK operation.
- static void [DSPI_HAL_SetModifiedTimingFormatCmd](#) (uint32_t baseAddr, bool enable)
Configures the DSPI to enable modified timing format.
- static void [DSPI_HAL_SetPcsStrobeCmd](#) (uint32_t baseAddr, bool enable)
Configures the DSPI peripheral chip select strobe enable.
- static void [DSPI_HAL_SetRxFifoOverwriteCmd](#) (uint32_t baseAddr, bool enable)
Configures the DSPI received FIFO overflow overwrite enable.
- void [DSPI_HAL_SetPcsPolarityMode](#) (uint32_t baseAddr, [dspi_which_pcs_config_t](#) pcs, [dspi_pcs_polarity_config_t](#) activeLowOrHigh)
Configures the DSPI peripheral chip select polarity.
- void [DSPI_HAL_SetFifoCmd](#) (uint32_t baseAddr, bool enableTxFifo, bool enableRxFifo)
Enables (or disables) the DSPI FIFOs.
- void [DSPI_HAL_SetFlushFifoCmd](#) (uint32_t baseAddr, bool enableFlushTxFifo, bool enableFlushRxFifo)
Flushes the DSPI FIFOs.
- static void [DSPI_HAL_SetDataInSamplepointMode](#) (uint32_t baseAddr, [dspi_master_sample_point_t](#) samplePnt)
Configures the time when the DSPI master samples SIN in the Modified Transfer Format.
- static void [DSPI_HAL_StartTransfer](#) (uint32_t baseAddr)
Starts the DSPI transfers, clears HALT bit in MCR.
- static void [DSPI_HAL_StopTransfer](#) (uint32_t baseAddr)
Stops (halts) DSPI transfers, sets HALT bit in MCR.
- [dspi_status_t](#) [DSPI_HAL_SetDataFormat](#) (uint32_t baseAddr, [dspi_ctar_selection_t](#) whichCtar, const [dspi_data_format_config_t](#) *config)
Configures the data format for a particular CTAR.
- void [DSPI_HAL_SetDelay](#) (uint32_t baseAddr, [dspi_ctar_selection_t](#) whichCtar, uint32_t prescaler, uint32_t scaler, [dspi_delay_type_t](#) whichDelay)
Manually configures the delay prescaler and scaler for a particular CTAR.
- uint32_t [DSPI_HAL_CalculateDelay](#) (uint32_t baseAddr, [dspi_ctar_selection_t](#) whichCtar, [dspi_delay_type_t](#) whichDelay, uint32_t sourceClockInHz, uint32_t delayInNanoSec)

Calculates the delay prescaler and scaler based on the desired delay input in nanoseconds.

Low power

- static void **DSPI_HAL_SetDozemodeCmd** (uint32_t baseAddr, bool enable)
Configures the DSPI operation during doze mode.

Interrupts

- void **DSPI_HAL_SetTxFifoFillDmaIntMode** (uint32_t baseAddr, **dspi_dma_or_int_mode_t** mode, bool enable)
Configures the DSPI Tx FIFO fill request to generate DMA or interrupt requests.
- void **DSPI_HAL_SetRxFifoDrainDmaIntMode** (uint32_t baseAddr, **dspi_dma_or_int_mode_t** mode, bool enable)
Configures the DSPI Rx FIFO Drain request to generate DMA or interrupt requests.
- void **DSPI_HAL_SetIntMode** (uint32_t baseAddr, **dspi_status_and_interrupt_request_t** interruptSrc, bool enable)
Configures the DSPI interrupts.
- static bool **DSPI_HAL_GetIntMode** (uint32_t baseAddr, **dspi_status_and_interrupt_request_t** interruptSrc)
Gets DSPI interrupt configuration, returns if interrupt request is enabled or disabled.

Status

- static bool **DSPI_HAL_GetStatusFlag** (uint32_t baseAddr, **dspi_status_and_interrupt_request_t** statusFlag)
Gets the DSPI status flag state.
- static void **DSPI_HAL_ClearStatusFlag** (uint32_t baseAddr, **dspi_status_and_interrupt_request_t** statusFlag)
Clears the DSPI status flag.
- static uint32_t **DSPI_HAL_GetFifoCountOrPtr** (uint32_t baseAddr, **dspi_fifo_counter_pointer_t** desiredParameter)
Gets the DSPI FIFO counter or pointer.

Data transfer

- static uint32_t **DSPI_HAL_ReadData** (uint32_t baseAddr)
Reads data from the data buffer.
- static void **DSPI_HAL_WriteDataSlavemode** (uint32_t baseAddr, uint32_t data)
Writes data into the data buffer, slave mode.
- void **DSPI_HAL_WriteDataMastermode** (uint32_t baseAddr, **dspi_command_config_t** *command, uint16_t data)
Writes data into the data buffer, master mode.

DSPI HAL driver

- void [DSPI_HAL_WriteDataMastermodeBlocking](#) (uint32_t baseAddr, [dsPIC_Command_Config_t](#) *command, uint16_t data)
Writes data into the data buffer, master mode and waits till complete to return.
- static uint32_t [DSPI_HAL_GetTransferCount](#) (uint32_t baseAddr)
Gets the transfer count.
- static void [DSPI_HAL_PresetTransferCount](#) (uint32_t baseAddr, uint16_t presetValue)
Pre-sets the transfer count.

Debug

- uint32_t [DSPI_HAL_GetFifoData](#) (uint32_t baseAddr, [dsPIC_Fifo_t](#) whichFifo, uint32_t whichFifoEntry)
Reads FIFO registers for debug purposes.
- static void [DSPI_HAL_SetHaltInDebugmodeCmd](#) (uint32_t baseAddr, bool enable)
Configures the DSPI to halt during debug mode.

8.1.1 Data Structure Documentation

8.1.1.1 struct dsPIC_data_format_config_t

This structure contains the data format settings. These settings apply to a specific CTARn register, which the user must provide in this structure.

Data Fields

- uint32_t [bitsPerFrame](#)
Bits per frame, minimum 4, maximum 16 (master), 32 (slave)
- [dsPIC_clock_polarity_t](#) [clkPolarity](#)
Active high or low clock polarity.
- [dsPIC_clock_phase_t](#) [clkPhase](#)
Clock phase setting to change and capture data.
- [dsPIC_shift_direction_t](#) [direction](#)
MSB or LSB data shift direction This setting relevant only in master mode and can be ignored in slave mode.

8.1.1.2 struct dsPIC_slave_config_t

Use an instance of this structure with the [DSPI_HAL_SlaveInit\(\)](#) to configure the most common settings of the DSPI peripheral in slave mode with a single function call.

Data Fields

- bool [isEnabled](#)
Set to true to enable the DSPI peripheral.

- **dspi_data_format_config_t dataConfig**
Data format configuration structure.
- **bool isTxFifoDisabled**
Disable(1) or Enable(0) Tx FIFO.
- **bool isRxFifoDisabled**
Disable(1) or Enable(0) Rx FIFO.

8.1.1.2.0.13 Field Documentation

8.1.1.2.0.13.1 bool dspi_slave_config_t::isEnabled

8.1.1.3 struct dspi_baud_rate_divisors_t

Note: These settings are relevant only in master mode. This structure contains the baud rate divisor settings, which provides the user with the option to explicitly set these baud rate divisors. In addition, the user must also set the CTARn register with the divisor settings.

Data Fields

- **bool doubleBaudRate**
Double Baud rate parameter setting.
- **uint32_t prescaleDivisor**
Baud Rate Pre-scalar parameter setting.
- **uint32_t baudRateDivisor**
Baud Rate scalar parameter setting.

8.1.1.4 struct dspi_command_config_t

Note: This structure is used with the PUSHR register, which provides the means to write to the Tx FIFO. Data written to this register is transferred to the Tx FIFO. Eight or sixteen-bit write accesses to the PUSHR transfer all 32 register bits to the Tx FIFO. The register structure is different in master and slave modes. In master mode, the register provides 16-bit command and 16-bit data to the Tx FIFO. In slave mode all 32 register bits can be used as data, supporting up to 32-bit SPI frame operation.

Data Fields

- **bool isChipSelectContinuous**
Option to enable the continuous assertion of chip select between transfers.
- **dspi_ctar_selection_t whichCtar**
The desired Clock and Transfer Attributes Register (CTAR) to use for CTAS.
- **dspi_which_pcs_config_t whichPcs**
The desired PCS signal to use for the data transfer.
- **bool isEndOfQueue**
Signals that the current transfer is the last in the queue.
- **bool clearTransferCount**
Clears SPI_TCNT field; cleared before transmission starts.

8.1.2 Enumeration Type Documentation

8.1.2.1 enum dspi_status_t

Enumerator

kStatus_DSPI_SlaveTxUnderrun DSPI Slave Tx Under run error.

kStatus_DSPI_SlaveRxOverrun DSPI Slave Rx Overrun error.

kStatus_DSPI_Timeout DSPI transfer timed out.

kStatus_DSPI_Busy DSPI instance is already busy performing a transfer.

kStatus_DSPI_NoTransferInProgress Attempt to abort a transfer when no transfer was in progress.

kStatus_DSPI_InvalidBitCount bits-per-frame value not valid

kStatus_DSPI_InvalidInstanceNumber DSPI instance number does not match current count.

kStatus_DSPI_OutOfRange DSPI out-of-range error used in slave callback.

8.1.2.2 enum dspi_master_slave_mode_t

Enumerator

kDspiMaster DSPI peripheral operates in master mode.

kDspiSlave DSPI peripheral operates in slave mode.

8.1.2.3 enum dspi_clock_polarity_t

Enumerator

kDspiClockPolarity_ActiveHigh Active-high DSPI clock (idles low)

kDspiClockPolarity_ActiveLow Active-low DSPI clock (idles high)

8.1.2.4 enum dspi_clock_phase_t

Enumerator

kDspiClockPhase_FirstEdge Data is captured on the leading edge of the SCK and changed on the following edge.

kDspiClockPhase_SecondEdge Data is changed on the leading edge of the SCK and captured on the following edge.

8.1.2.5 enum dspi_shift_direction_t

Enumerator

kDspiMsbFirst Data transfers start with most significant bit.

kDspiLsbFirst Data transfers start with least significant bit.

8.1.2.6 enum dspi_ctar_selection_t

Enumerator

kDspiCtar0 CTAR0 selection option for master or slave mode.

kDspiCtar1 CTAR1 selection option for master mode only.

8.1.2.7 enum dspi_pcs_polarity_config_t

Enumerator

kDspiPcs_ActiveHigh PCS Active High (idles low)

kDspiPcs_ActiveLow PCS Active Low (idles high)

8.1.2.8 enum dspi_which_pcs_config_t

Enumerator

kDspiPcs0 PCS[0].

kDspiPcs1 PCS[1].

kDspiPcs2 PCS[2].

kDspiPcs3 PCS[3].

kDspiPcs4 PCS[4].

kDspiPcs5 PCS[5].

8.1.2.9 enum dspi_master_sample_point_t

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

Enumerator

kDspiSckToSin_0Clock 0 system clocks between SCK edge and SIN sample

kDspiSckToSin_1Clock 1 system clock between SCK edge and SIN sample

kDspiSckToSin_2Clock 2 system clocks between SCK edge and SIN sample

DSPI HAL driver

8.1.2.10 enum dspi_fifo_t

Enumerator

kDspiTxFifo DSPI Tx FIFO.

kDspiRxFifo DSPI Rx FIFO.

8.1.2.11 enum dspi_dma_or_int_mode_t

Enumerator

kDspiGenerateIntReq Desired flag generates an Interrupt request.

kDspiGenerateDmaReq Desired flag generates a DMA request.

8.1.2.12 enum dspi_status_and_interrupt_request_t

Enumerator

kDspiTxComplete TCF status/interrupt enable.

kDspiTxAndRxStatus TXRXS status only, no interrupt.

kDspiEndOfQueue EOQF status/interrupt enable.

kDspiTxFifoUnderflow TFUF status/interrupt enable.

kDspiTxFifoFillRequest TFFF status/interrupt enable.

kDspiRxFifoOverflow RFOF status/interrupt enable.

kDspiRxFifoDrainRequest RFDF status/interrupt enable.

8.1.2.13 enum dspi_fifo_counter_pointer_t

Enumerator

kDspiRxFifoPointer Rx FIFO pointer.

kDspiRxFifoCounter Rx FIFO counter.

kDspiTxFifoPointer Tx FIFO pointer.

kDspiTxFifoCounter Tx FIFO counter.

8.1.2.14 enum dspi_delay_type_t

Enumerator

kDspiPcsToSck PCS-to-SCK delay.

kDspiLastSckToPcs Last SCK edge to PCS delay.

kDspiAfterTransfer Delay between transfers.

8.1.3 Function Documentation

8.1.3.1 void DSPI_HAL_Init(uint32_t *baseAddr*)

This function basically resets all of the DSPI registers to their default setting including disabling the module.

DSPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

8.1.3.2 static void DSPI_HAL_Enable (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

8.1.3.3 static void DSPI_HAL_Disable (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

8.1.3.4 uint32_t DSPI_HAL_SetBaudRate (uint32_t *baseAddr*, dspi_ctar_selection_t *whichCtar*, uint32_t *bitsPerSec*, uint32_t *sourceClockInHz*)

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

Parameters

<i>baseAddr</i>	Module base address
<i>whichCtar</i>	The desired Clock and Transfer Attributes Register (CTAR) of the type dspi_ctar_selection_t
<i>bitsPerSec</i>	The desired baud rate in bits per second
<i>sourceClockInHz</i>	Module source input clock in Hertz

Returns

The actual calculated baud rate

8.1.3.5 void DSPI_HAL_SetBaudDivisors (uint32_t *baseAddr*, dspi_ctar_selection_t *whichCtar*, const dspi_baud_rate_divisors_t * *divisors*)

This function allows the caller to manually set the baud rate divisors in the event that these dividers are known and the caller does not wish to call the DSPI_HAL_SetBaudRate function.

DSPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
<i>whichCtar</i>	The desired Clock and Transfer Attributes Register (CTAR) of type <code>dspi_ctar_selection_t</code>
<i>divisors</i>	Pointer to a structure containing the user defined baud rate divisor settings

8.1.3.6 static void DSPI_HAL_SetMasterSlaveMode (uint32_t *baseAddr*, dspi_master_slave_mode_t *mode*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>mode</i>	Mode setting (master or slave) of type <code>dspi_master_slave_mode_t</code>

8.1.3.7 static bool DSPI_HAL_IsMaster (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

Return values

<i>true</i>	The module is in master mode.
<i>false</i>	The module is in slave mode.

8.1.3.8 static void DSPI_HAL_SetContinuousSckCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enables (true) or disables(false) continuous SCK operation.

8.1.3.9 static void DSPI_HAL_SetModifiedTimingFormatCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enables (true) or disables(false) modified timing format.

8.1.3.10 static void DSPI_HAL_SetPcsStrobeCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Configures the PCS[5] to be the active-low PCS Strobe output.

PCS[5] is a special case that can be configured as an active low PCS strobe or as a Peripheral Chip Select in master mode. When configured as a strobe, it provides a signal to an external demultiplexer to decode PCS[0] to PCS[4] signals into as many as 128 glitch-free PCS signals.

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enable (true) PCS[5] to operate as the peripheral chip select (PCS) strobe If disable (false), PCS[5] operates as a peripheral chip select

8.1.3.11 static void DSPI_HAL_SetRx_fifoOverwriteCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

When enabled, this function allows incoming receive data to overwrite the existing data in the receive shift register when the Rx FIFO is full. Otherwise when disabled, the incoming data is ignored when the RX FIFO is full.

Parameters

<i>baseAddr</i>	Module base address.
<i>enable</i>	If enabled (true), allows incoming data to overwrite Rx FIFO contents when full, else incoming data is ignored.

8.1.3.12 void DSPI_HAL_SetPcsPolarityMode (uint32_t *baseAddr*, dspi_- which_pcs_config_t *pcs*, dspi_pcs_polarity_config_t *activeLowOrHigh*)

This function takes in the desired peripheral chip select (PCS) and it's corresponding desired polarity and configures the PCS signal to operate with the desired characteristic.

DSPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
<i>pcs</i>	The particular peripheral chip select (parameter value is of type dspi_which_pcs_config_t) for which we wish to apply the active high or active low characteristic.
<i>activeLowOrHigh</i>	The setting for either "active high, inactive low (0)" or "active low, inactive high(1)" of type dspi_pcs_polarity_config_t.

8.1.3.13 void DSPI_HAL_SetFifoCmd (uint32_t *baseAddr*, bool *enableTxFifo*, bool *enableRxFifo*)

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

Parameters

<i>baseAddr</i>	Module instance number
<i>enableTxFifo</i>	Disables (false) the TX FIFO, else enables (true) the TX FIFO
<i>enableRxFifo</i>	Disables (false) the RX FIFO, else enables (true) the RX FIFO

8.1.3.14 void DSPI_HAL_SetFlushFifoCmd (uint32_t *baseAddr*, bool *enableFlushTxFifo*, bool *enableFlushRxFifo*)

Parameters

<i>baseAddr</i>	Module base address
<i>enableFlushTxFifo</i>	Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO
<i>enableFlushRxFifo</i>	Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO

8.1.3.15 static void DSPI_HAL_SetDataInSamplepointMode (uint32_t *baseAddr*, dspi_master_sample_point_t *samplePnt*) [inline], [static]

This function controls when the DSPI master samples SIN (data in) in the Modified Transfer Format. Note that this is valid only when the CPHA bit in the CTAR register is 0.

Parameters

<i>baseAddr</i>	Module base address
<i>samplePnt</i>	selects when the data in (SIN) is sampled, of type dspi_master_sample_point_t. This value selects either 0, 1, or 2 system clocks between the SCK edge and the SIN (data in) sample.

8.1.3.16 static void DSPI_HAL_StartTransfer (uint32_t *baseAddr*) [inline], [static]

This function call called whenever the module is ready to begin data transfers in either master or slave mode.

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

8.1.3.17 static void DSPI_HAL_StopTransfer (uint32_t *baseAddr*) [inline], [static]

This function call stops data transfers in either master or slave mode.

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

8.1.3.18 dspi_status_t DSPI_HAL_SetDataFormat (uint32_t *baseAddr*, dspi_ctar_selection_t *whichCtar*, const dspi_data_format_config_t * *config*)

This function configures the bits-per-frame, polarity, phase, and shift direction for a particular CTAR. An example use case is as follows:

```
dspi_data_format_config_t dataFormat;
dataFormat.bitsPerFrame = 16;
dataFormat.clkPolarity = kDspiClockPolarity_ActiveLow;
dataFormat.clkPhase = kDspiClockPhase_FirstEdge;
dataFormat.direction = kDspiMsbFirst;
DSPI_HAL_SetDataFormat(instance, kDspiCtar0, &dataFormat);
```

DSPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
<i>whichCtar</i>	The desired Clock and Transfer Attributes Register (CTAR) of type <code>dspi_ctar_selection_t</code> .
<i>config</i>	Pointer to structure containing user defined data format configuration settings.

Returns

An error code or `kStatus_DSPI_Success`

8.1.3.19 `void DSPI_HAL_SetDelay (uint32_t baseAddr, dspi_ctar_selection_t whichCtar, uint32_t prescaler, uint32_t scaler, dspi_delay_type_t whichDelay)`

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

These delay names are available in type `dspi_delay_type_t`.

The user passes which delay they want to configure along with the prescaler and scaler value. This allows the user to directly set the prescaler/scaler values if they have pre-calculated them or if they simply wish to manually increment either value.

Parameters

<i>baseAddr</i>	Module base address
<i>whichCtar</i>	The desired Clock and Transfer Attributes Register (CTAR) of type <code>dspi_ctar_selection_t</code> .
<i>prescaler</i>	The prescaler delay value (can be an integer 0, 1, 2, or 3).
<i>prescaler</i>	The scaler delay value (can be any integer between 0 to 15).
<i>whichDelay</i>	The desired delay to configure, must be of type <code>dspi_delay_type_t</code>

8.1.3.20 `uint32_t DSPI_HAL_CalculateDelay (uint32_t baseAddr, dspi_ctar_selection_t whichCtar, dspi_delay_type_t whichDelay, uint32_t sourceClockInHz, uint32_t delayInNanoSec)`

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

These delay names are available in type `dspi_delay_type_t`.

The user passes which delay they want to configure along with the desired delay value in nano-seconds. The function calculates the values needed for the prescaler and scaler and returning the actual calculated

delay as an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay will be returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Parameters

<i>baseAddr</i>	Module base address
<i>whichCtar</i>	The desired Clock and Transfer Attributes Register (CTAR) of type <code>dspi_ctar_selection_t</code> .
<i>whichDelay</i>	The desired delay to configure, must be of type <code>dspi_delay_type_t</code>
<i>sourceClockIn-Hz</i>	Module source input clock in Hertz
<i>delayInNano-Sec</i>	The desired delay value in nano-seconds.

Returns

The actual calculated delay value.

8.1.3.21 `static void DSPI_HAL_SetDozemodeCmd (uint32_t baseAddr, bool enable) [inline], [static]`

This function provides support for an externally controlled doze mode, power-saving, mechanism. When disabled, the doze mode has no effect on the DSPI, and when enabled, the Doze mode disables the DSPI.

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	If disabled (false), the doze mode has no effect on the DSPI, if enabled (true), the doze mode disables the DSPI.

8.1.3.22 `void DSPI_HAL_SetTxFifoFillDmaIntMode (uint32_t baseAddr, dspi_dma_or_int_mode_t mode, bool enable)`

This function configures the DSPI Tx FIFO Fill flag to generate either an interrupt or DMA request. The user passes in which request they'd like to generate of type `dspi_dma_or_int_mode_t` and whether or not they wish to enable this request. Note, when disabling the request, the request type is don't care.

```
DSPI_HAL_SetTxFifoFillDmaIntMode(baseAddr,
    kDspiGenerateDmaReq, true); <- to enable DMA
DSPI_HAL_SetTxFifoFillDmaIntMode(baseAddr,
    kDspiGenerateIntReq, true); <- to enable Interrupt
```

DSPI HAL driver

```
DSPI_HAL_SetTxFifoFillDmaIntMode(baseAddr,  
    kDspiGenerateIntReq, false); <- to disable
```

Parameters

<i>baseAddr</i>	Module base address
<i>mode</i>	Configures the DSPI Tx FIFO Fill to generate an interrupt or DMA request
<i>enable</i>	Enable (true) or disable (false) the DSPI Tx FIFO Fill flag to generate requests

8.1.3.23 void DSPI_HAL_SetRxFifoDrainDmaIntMode (uint32_t *baseAddr*, dspi_dma_or_int_mode_t *mode*, bool *enable*)

This function configures the DSPI Rx FIFO Drain flag to generate either an interrupt or a DMA request. The user passes in which request they'd like to generate of type dspi_dma_or_int_mode_t and whether or not they wish to enable this request. Note, when disabling the request, the request type is don't care.

```
DSPI_HAL_SetRxFifoDrainDmaIntMode(baseAddr,
    kDspiGenerateDmaReq, true); <- to enable DMA
DSPI_HAL_SetRxFifoDrainDmaIntMode(baseAddr,
    kDspiGenerateIntReq, true); <- to enable Interrupt
DSPI_HAL_SetRxFifoDrainDmaIntMode(baseAddr,
    kDspiGenerateIntReq, false); <- to disable
```

Parameters

<i>baseAddr</i>	Module base address
<i>mode</i>	Configures the Rx FIFO Drain to generate an interrupt or DMA request
<i>enable</i>	Enable (true) or disable (false) the Rx FIFO Drain flag to generate requests

8.1.3.24 void DSPI_HAL_SetIntMode (uint32_t *baseAddr*, dspi_status_and_interrupt_request_t *interruptSrc*, bool *enable*)

This function configures the various interrupt sources of the DSPI. The parameters are baseAddr, interrupt source, and enable/disable setting. The interrupt source is a typedef enumeration whose value is the bit position of the interrupt source setting within the RSER register. In the DSPI, all interrupt configuration settings are in one register. The typedef enum equates each interrupt source to the bit position defined in the device header file. The function uses these bit positions in its algorithm to enable/disable the interrupt source, where interrupt source is the dspi_status_and_interrupt_request_t type. Note, for Tx FIFO Fill and Rx FIFO Drain requests, use the functions: DSPI_HAL_SetTxFifoFillDmaIntMode and DSPI_HAL_SetRxFifoDrainDmaIntMode respectively as these requests can generate either an interrupt or DMA request.

```
DSPI_HAL_SetIntMode(baseAddr, kDspiTxComplete, true); <- example use-case
```

DSPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
<i>interruptSrc</i>	The interrupt source, of type dspi_status_and_interrupt_request_t
<i>enable</i>	Enable (true) or disable (false) the interrupt source to generate requests

8.1.3.25 static bool DSPI_HAL_GetIntMode (uint32_t *baseAddr*, dspi_status_and_interrupt_request_t *interruptSrc*) [inline], [static]

This function returns the requested interrupt source setting (enabled or disabled, of type bool). The parameters to pass in are baseAddr and interrupt source. It utilizes the same enumeration definitions for the interrupt sources as described in the "interrupt configuration" function. The function uses these bit positions in its algorithm to obtain the desired interrupt source setting. Note, for Tx FIFO Fill and Rx FIFO Drain requests, this returns whether or not their requests are enabled.

```
getInterruptSetting = DSPI_HAL_GetIntMode(baseAddr,  
    kDspiTxComplete);
```

Parameters

<i>baseAddr</i>	Module base address
<i>interruptSrc</i>	The interrupt source, of type dspi_status_and_interrupt_request_t

Returns

Configuration of interrupt request: enable (true) or disable (false).

8.1.3.26 static bool DSPI_HAL_GetStatusFlag (uint32_t *baseAddr*, dspi_status_and_interrupt_request_t *statusFlag*) [inline], [static]

The status flag is defined in the same enumeration as the interrupt source enable because the bit position of the interrupt source and corresponding status flag are the same in the RSER and SR registers. The function uses these bit positions in its algorithm to obtain the desired flag state, similar to the dspi_get_interrupt_config function.

```
getStatus = DSPI_HAL_GetStatusFlag(baseAddr,  
    kDspiTxComplete);
```

Parameters

<i>baseAddr</i>	Module base address
<i>statusFlag</i>	The status flag, of type <code>dspi_status_and_interrupt_request_t</code>

Returns

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

8.1.3.27 `static void DSPI_HAL_ClearStatusFlag (uint32_t baseAddr, dspi_status_and_interrupt_request_t statusFlag) [inline], [static]`

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

```
DSPI_HAL_ClearStatusFlag(baseAddr, kDspiTxComplete);
```

Parameters

<i>baseAddr</i>	Module base address
<i>statusFlag</i>	The status flag, of type <code>dspi_status_and_interrupt_request_t</code>

8.1.3.28 `static uint32_t DSPI_HAL_GetFifoCountOrPtr (uint32_t baseAddr, dspi_fifo_counter_pointer_t desiredParameter) [inline], [static]`

This function returns the number of entries or the next pointer in the Tx or Rx FIFO. The parameters to pass in are the baseAddr and either the Tx or Rx FIFO counter or a pointer. The latter is an enumeration type defined as the bitmask of those particular bit fields found in the device header file. Example usage:

```
DSPI_HAL_GetFifoCountOrPtr(baseAddr,
    kDspiRxFifoCounter);
```

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

DSPI HAL driver

<i>desired-Parameter</i>	Desired parameter to obtain, of type dspi_fifo_counter_pointer_t
--------------------------	--

8.1.3.29 static uint32_t DSPI_HAL_ReadData (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

8.1.3.30 static void DSPI_HAL_WriteDataSlavemode (uint32_t *baseAddr*, uint32_t *data*) [inline], [static]

In slave mode, up to 32-bit words may be written.

Parameters

<i>baseAddr</i>	Module base address
<i>data</i>	The data to send

8.1.3.31 void DSPI_HAL_WriteDataMastermode (uint32_t *baseAddr*, dspi_command_config_t * *command*, uint16_t *data*)

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

```
dspi_command_config_t commandConfig;
commandConfig.isChipSelectContinuous = true;
commandConfig.whichCtar = kDspiCtar0;
commandConfig.whichPcs = kDspiPcs1;
commandConfig.clearTransferCount = false;
commandConfig.isEndOfQueue = false;
DSPI_HAL_WriteDataMastermode(baseAddr, &commandConfig, dataWord);
```

Parameters

<i>baseAddr</i>	Module base address
<i>command</i>	Pointer to command structure
<i>data</i>	The data word to be sent

8.1.3.32 void DSPI_HAL_WriteDataMastermodeBlocking (*uint32_t baseAddr*, *dspi_command_config_t * command*, *uint16_t data*)

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

```
dspi_command_config_t commandConfig;
commandConfig.isChipSelectContinuous = true;
commandConfig.whichCtar = kDspiCtar0;
commandConfig.whichPcs = kDspiPcs1;
commandConfig.clearTransferCount = false;
commandConfig.isEndOfQueue = false;
DSPI_HAL_WriteDataMastermode(baseAddr, &commandConfig, dataWord);
```

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

Parameters

<i>baseAddr</i>	Module base address
<i>command</i>	Pointer to command structure
<i>data</i>	The data word to be sent

8.1.3.33 static uint32_t DSPI_HAL_GetTransferCount (*uint32_t baseAddr*) [inline], [static]

This function returns the current value of the DSPI Transfer Count Register.

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

DSPI HAL driver

Returns

The current transfer count

8.1.3.34 static void DSPI_HAL_PresetTransferCount (*uint32_t baseAddr*, *uint16_t presetValue*) [inline], [static]

This function allows the caller to pre-set the DSI Transfer Count Register to a desired value up to 65535; Incrementing past this resets the counter back to 0.

Parameters

<i>baseAddr</i>	Module base address
<i>presetValue</i>	The desired pre-set value for the transfer counter

8.1.3.35 uint32_t DSPI_HAL_GetFifoData (*uint32_t baseAddr*, *dspi_fifo_t whichFifo*, *uint32_t whichFifoEntry*)

Parameters

<i>baseAddr</i>	Module base address
<i>whichFifo</i>	Selects Tx or Rx FIFO, of type <i>dspi_fifo_t</i> .
<i>whichFifoEntry</i>	Selects which FIFO entry to read: 0, 1, 2, or 3.

Returns

The desired FIFO register contents

8.1.3.36 static void DSPI_HAL_SetHaltInDebugmodeCmd (*uint32_t baseAddr*, *bool enable*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enables (true) debug mode to halt transfers, else disable to not halt transfer in debug mode.

8.2 DSPI Master Driver

This chapter describes the programming interface of the DSPI master mode Peripheral driver.

Data Structures

- struct [dspi_device_t](#)
Data structure containing information about a device on the SPI bus. [More...](#)
- struct [dspi_master_state_t](#)
Runtime state structure for the DSPI master driver. [More...](#)
- struct [dspi_master_user_config_t](#)
The user configuration structure for the DSPI master driver. [More...](#)

Initialization and shutdown

- [dspi_status_t DSPI_DRV_MasterInit](#) (uint32_t instance, [dspi_master_state_t](#) *dspiState, const [dspi_master_user_config_t](#) *userConfig)
Initializes a DSPI instance for master mode operation.
- [dspi_status_t DSPI_DRV_MasterInitDma](#) (uint32_t instance, [dspi_master_state_t](#) *dspiState, const [dspi_master_user_config_t](#) *userConfig, [edma_software_tcd_t](#) *stcdSrc2CmdDataLast)
Initializes a DSPI instance for master mode operation with DMA support.
- void [DSPI_DRV_MasterDeinit](#) (uint32_t instance)
Shuts down a DSPI instance.
- [dspi_status_t DSPI_DRV_MasterSetDelay](#) (uint32_t instance, [dspi_delay_type_t](#) whichDelay, uint32_t delayInNanoSec, uint32_t *calculatedDelay)
Configures the DSPI master mode bus timing delay options.

Bus configuration

- [dspi_status_t DSPI_DRV_MasterConfigureBus](#) (uint32_t instance, const [dspi_device_t](#) *device, uint32_t *calculatedBaudRate)
Configures the DSPI port physical parameters to access a device on the bus.

Blocking transfers

- [dspi_status_t DSPI_DRV_MasterTransferDataBlocking](#) (uint32_t instance, const [dspi_device_t](#) *restrict device, const uint8_t *sendBuffer, uint8_t *receiveBuffer, size_t transferByteCount, uint32_t timeout)
Performs a blocking SPI master mode transfer.

Non-blocking transfers

- [dspi_status_t DSPI_DRV_MasterTransferData](#) (uint32_t instance, const [dspi_device_t](#) *restrict device, const uint8_t *sendBuffer, uint8_t *receiveBuffer, size_t transferByteCount)

DSPI Master Driver

- *Performs a non-blocking SPI master mode transfer.*
• `dspi_status_t DSPI_DRV_MasterGetTransferStatus (uint32_t instance, uint32_t *framesTransferred)`
Returns whether the previous transfer is finished.
- `dspi_status_t DSPI_DRV_MasterAbortTransfer (uint32_t instance)`
Terminates an asynchronous transfer early.

8.2.0.37 DSPI Master Driver

Baud rate in bits per second.*/ `dspi_data_format_config_t` dataBusConfig; /* data format config struct*/ }
`dspi_device_t`;

Overview

The DSPI master mode peripheral driver transfers data to and from external devices on the DSPI bus in master mode. It supports transferring buffers of data with a single function call.

Run-time state structures

The DSPI master driver uses a run-time state structure, `dspi_master_state_t`, to track the ongoing data transfer. The structure holds data that the DSPI master peripheral driver uses to communicate between the transfer function and the interrupt handler. The interrupt handler also uses this information to keep track of its progress. The user is only responsible to pass the memory for this run-time state structure and the DSPI master driver fills out the members.

User configuration structures

The DSPI master driver uses instances of the user configuration structure, `dspi_master_user_config_t`, for the DSPI master driver. For this reason, the user can configure the most common settings of the DSPI peripheral with a single function call.

Device structures

The DSPI master driver uses instances of the `dspi_device_t` structure to represent the SPI bus configuration required to communicate to an external device that is connected to the bus.

The device structure can be passed to the data transfer functions, and the bus is reconfigured before the transfer is started. Alternatively, the user can configure the SPI bus for a device manually. For instance, if there is only one device connected to the bus, the user might configure it only once.

Initialization

To initialize the DSPI master driver, call the [DSPI_DRV_MasterInit\(\)](#) function and pass the instance number of the DSPI peripheral, the user configuration, [dspi_master_user_config_t](#), a pointer to the variable for retrieving the calculated baud rate (this can be a NULL), and memory allocation for the run-time state structure used by the master driver to keep track of data transfers. Following the DSPI master initialization is the DSPI master configure bus. While the [DSPI_DRV_MasterInit\(\)](#) function initializes the DSPI peripheral, the [DSPI_DRV_MasterConfigureBus\(\)](#) function configures the SPI bus parameters such as bits/frame, clock characteristics, data shift direction, baud rate, and desired chip select. More details to follow. For example, to use the DSPI1, pass a value of 1 to the initialization function. This is an example for the other parameters.

Example code to initialize and configure the driver:

```
/* Set up and init the master */
uint32_t masterInstance = 1;
dspi_master_state_t dspiMasterState;
uint32_t calculatedBaudRate;

/* configure the members of the user config */
dspi_master_user_config_t userConfig;
userConfig.isChipSelectContinuous = false;
userConfig.isSckContinuous = false;
userConfig.pcsPolarity = kDspiPcs_ActiveLow;
userConfig.whichCtar = kDspiCtar0;
userConfig.whichPcs = kDspiPcs1;

/* init the DSPI module */
DSPI_DRV_MasterInit(masterInstance, &dspiMasterState, &userConfig, &calculatedBaudRate);

// Define bus configuration.
dspi_device_t spiDevice;
spiDevice.dataBusConfig.bitsPerFrame = 16;
spiDevice.dataBusConfig.clkPhase = kDspiClockPhase_FirstEdge;
spiDevice.dataBusConfig.clkPolarity = kDspiClockPolarity_ActiveHigh
    ;
spiDevice.dataBusConfig.direction = kDspiMsbFirst;
spiDevice.bitsPerSec = 500000;

/* configure the SPI bus */
DSPI_DRV_MasterConfigureBus(masterInstance, &spiDevice, &calculatedBaudRate);
```

Additionally, the DSPI supports DMA transfers, but only for DSPI instance 0. To use the DSPI with DMA, call an alternate initialization function, [DSPI_DRV_MasterInitDma](#). The user still needs to also call the [DSPI_DRV_MasterConfigureBus\(\)](#) function.

```
/* Function prototype */
dspi_status_t DSPI_DRV_MasterInitDma(uint32_t instance,
                                      dspi_master_state_t * dspiState,
                                      const dspi_master_user_config_t * userConfig,
                                      edma_software_tcd_t * stcdSrc2CmdDataLast);
```

This function requires one additional parameter, [stcdSrc2CmdDataLast](#), a pointer to a structure of the type [edma_software_tcd_t](#). This pointer needs to be aligned to a 32-byte boundary.

In addition to initialization of the DSPI interface, it is also necessary to configure the SPI to match the parameters needed by the peripheral device as mentioned above.

DSPI Master Driver

```
dspi_status_t DSPI_DRV_MasterConfigureBus(uint32_t instance
                                         const dsPIC_Device_t * device,
                                         uint32_t * calculatedBaudRate);
```

The user passes in a device structure that represents the characteristics of the SPI bus including the desired baud rate. Note that, in some cases, the exact baud rate cannot be achieved. Instead, the closest matching baud rate is returned via the calculatedBaudRate parameter. Note that this parameter never exceeds the desired baud rate. An example device structure usage is as follows:

```
typedef struct DSPIDevice {
    uint32_t bitsPerSec;
    dsPIC_Clock_Polarity_t clkPolarity;
    dsPIC_Clock_Phase_t clkPhase;
    dsPIC_Shift_Direction_t direction;
} dsPIC_Data_Format_Config_t;

/* example usage */
dsPIC_Device_t spiDevice;
spiDevice.dataBusConfig.bitsPerFrame = bitCount;
spiDevice.dataBusConfig.clkPhase = kDsPIClockPhase_FirstEdge;
spiDevice.dataBusConfig.clkPolarity =
    kDsPIClockPolarity_ActiveHigh;
spiDevice.dataBusConfig.direction = kDsPICmsbFirst;
spiDevice.bitsPerSec = baudRate;
```

Transfers

The driver supports two different modes for transferring data: blocking and non-blocking (async). The [DSPI_DRV_MasterTransferDataBlocking\(\)](#) is the blocking transfer function. The [DSPI_DRV_MasterTransferData\(\)](#) is a non-blocking function.

Example of a blocking transfer:

```
/* Function prototype */
status_t DSPI_DRV_MasterTransferDataBlocking(masterInstance,
                                              const dsPIC_Device_t * restrict device,
                                              const uint8_t * sendBuffer,
                                              uint8_t * receiveBuffer,
                                              uint32_t transferByteCount,
                                              uint32_t timeout);

/* Example function call */
DSPI_DRV_MasterTransferDataBlocking(masterInstance, NULL,
                                     s_dspisourceBuffer[masterInstance], s_dspisinkBuffer[masterInstance], 32, 1);
```

Example of a non-blocking transfer:

```
/* Function prototype */
status_t DSPI_DRV_MasterTransferData(masterInstance,
                                      const dsPIC_Device_t * restrict device,
                                      const uint8_t * sendBuffer,
                                      uint8_t * receiveBuffer,
                                      uint32_t transferByteCount);

/* Example function call */
DSPI_DRV_MasterTransferData(masterInstance, NULL, s_dspisourceBuffer[
    masterInstance], s_dspisinkBuffer[masterInstance], 32);
```

```
/* For non-blocking/async transfers, need to check back to get transfer status, for example */
/* Where framesXfer returns the number of frames transferred */
DSPI_DRV_MasterGetTransferStatus(masterInstance, &framesXfer);
```

To abort a transfer, simply call:

```
dspi_status_t DSPI_DRV_MasterAbortTransfer(masterInstance);
```

De-initialization

To de-initialize or shut down the DSPI module, call the function:

```
void DSPI_DRV_MasterDeinit(masterInstance);
```

8.2.1 Data Structure Documentation

8.2.1.1 struct dspl_device_t

The user must fill out these members to set up the DSPI master to properly communicate with the SPI device.

Data Fields

- uint32_t bitsPerSec
Baud rate in bits per second.

8.2.1.1.0.14 Field Documentation

8.2.1.1.0.14.1 uint32_t dspl_device_t::bitsPerSec

8.2.1.2 struct dspl_master_state_t

This structure holds data that is used by the DSPI master peripheral driver to communicate between the transfer function and the interrupt handler. The interrupt handler also uses this information to keep track of its progress. The user must pass the memory for this run-time state structure and the DSPI master driver fills out the members.

Data Fields

- dspl_ctar_selection_t whichCtar
Desired Clock and Transfer Attributes Register (CTAR)
- uint32_t bitsPerFrame
Desired number of bits per frame.
- dspl_which_pcs_config_t whichPcs

DSPI Master Driver

- *Desired Peripheral Chip Select (pcs)*
 - `bool isChipSelectContinuous`
Option to enable the continuous assertion of chip select between transfers.
 - `uint32_t dspiSourceClock`
Module source clock.
 - `volatile bool isTransferInProgress`
True if there is an active transfer.
 - `bool isTransferAsync`
Whether the transfer is asynchronous (needed in IRQ).
 - `const uint8_t *restrict sendBuffer`
The buffer from which transmitted bytes are taken.
 - `uint8_t *restrict receiveBuffer`
The buffer into which received bytes are placed.
 - `volatile size_t remainingSendByteCount`
Number of bytes remaining to send.
 - `volatile size_t remainingReceiveByteCount`
Number of bytes remaining to receive.
 - `semaphore_t irqSync`
Used to wait for ISR to complete its business.
 - `bool useDma`
User option to invoke usage of DMA.
 - `edma_chn_state_t dmaCmdData2Fifo`
Structure definition for the eDMA channel.
 - `edma_chn_state_t dmaSrc2CmdData`
Structure definition for the eDMA channel.
 - `edma_chn_state_t dmaFifo2Receive`
Structure definition for the eDMA channel.
 - `edma_software_tcd_t * stcdSrc2CmdDataLast`
Pointer to SW TCD in memory.

8.2.1.2.0.15 Field Documentation

8.2.1.2.0.15.1 `volatile bool dspi_master_state_t::isTransferInProgress`

8.2.1.2.0.15.2 `bool dspi_master_state_t::isTransferAsync`

8.2.1.2.0.15.3 `const uint8_t* restrict dspi_master_state_t::sendBuffer`

8.2.1.2.0.15.4 `uint8_t* restrict dspi_master_state_t::receiveBuffer`

8.2.1.2.0.15.5 `volatile size_t dspi_master_state_t::remainingSendByteCount`

8.2.1.2.0.15.6 `volatile size_t dspi_master_state_t::remainingReceiveByteCount`

8.2.1.2.0.15.7 `semaphore_t dspi_master_state_t::irqSync`

8.2.1.3 `struct dspi_master_user_config_t`

Use an instance of this structure with the `DSPI_DRV_MasterInit()`. This allows the user to configure the most common settings of the DSPI peripheral with a single function call.

Data Fields

- **dspi_ctar_selection_t whichCtar**
Desired Clock and Transfer Attributes Register(CTAR)
- bool **isSckContinuous**
Disable or Enable continuous SCK operation.
- bool **isChipSelectContinuous**
Option to enable the continuous assertion of chip select between transfers.
- **dspi_which_pcs_config_t whichPcs**
Desired Peripheral Chip Select (pcs)
- **dspi_pcs_polarity_config_t pcsPolarity**
Peripheral Chip Select (pcs) polarity setting.

8.2.1.3.0.16 Field Documentation

8.2.1.3.0.16.1 **dspi_pcs_polarity_config_t dspi_master_user_config_t::pcsPolarity**

8.2.2 Function Documentation

8.2.2.1 **dspi_status_t DSPI_DRV_MasterInit (uint32_t instance, dspi_master_state_t *dsSpiState, const dspi_master_user_config_t * userConfig)**

This function uses a CPU interrupt driven method for transferring data. This function initializes the run-time state structure to track the ongoing transfers, ungates the clock to the DSPI module, resets the DSPI module, initializes the module to user defined settings and default settings, configures the IRQ state structure, enables the module-level interrupt to the core, and enables the DSPI module. The CTAR parameter is special in that it allows the user to have different SPI devices connected to the same DSPI module instance in addition to different peripheral chip selects. Each CTAR contains the bus attributes associated with that particular SPI device. For most use cases where only one SPI device is connected per DSPI module instance, use CTAR0. This is an example to set up the **dspi_master_state_t** and the **dspi_master_user_config_t** parameters and to call the DSPI_DRV_MasterInit function by passing in these parameters:

```
dspi_master_state_t dspiMasterState; <- the user simply allocates memory for this struct
uint32_t calculatedBaudRate;
dspi_master_user_config_t userConfig; <- the user fills out members for this
    struct
userConfig.isChipSelectContinuous = false;
userConfig.isSckContinuous = false;
userConfig.pcsPolarity = kDspiPcs_ActiveLow;
userConfig.whichCtar = kDspiCtar0;
userConfig.whichPcs = kDspiPcs0;
DSPI_DRV_MasterInit(masterInstance, &dspiMasterState, &userConfig);
```

Parameters

DSPI Master Driver

<i>instance</i>	The instance number of the DSPI peripheral.
<i>dspiState</i>	The pointer to the DSPI master driver state structure. The user must pass the memory for this run-time state structure and the DSPI master driver fills out the members. This run-time state structure keeps track of the transfer in progress.
<i>userConfig</i>	The dspi_master_user_config_t user configuration structure. The user must fill out the members of this structure and pass the pointer of this struct into the function.

Returns

An error code or kStatus_DSPI_Success.

8.2.2.2 **dspi_status_t DSPI_DRV_MasterInitDma (uint32_t *instance*, dspi_master_state_t * *dspiState*, const dspi_master_user_config_t * *userConfig*, edma_software_tcd_t * *stcdSrc2CmdDataLast*)**

This function is exactly like the DSPI_DRV_MasterInit function but in addition, adds DMA support. If the user desires to use DMA based transfers, then the user should use this function call instead of the DSPI_DRV_MasterInit function call. Like the DSPI_DRV_MasterInit, this function initializes the run-time state structure to track the ongoing transfers, ungates the clock to the DSPI module, resets the DSPI module, initializes the module to user defined settings and default settings, configures the IRQ state structure, enables the module-level interrupt to the core, and enables the DSPI module. The CTAR parameter is special in that it allows the user to have different SPI devices connected to the same DSPI module instance in addition to different peripheral chip selects. Each CTAR contains the bus attributes associated with that particular SPI device. For most use cases where only one SPI device is connected per DSPI module instance, use CTAR0. This is an example to set up the [dspi_master_state_t](#) and the [dspi_master_user_config_t](#) parameters and to call the DSPI_DRV_MasterInit function by passing in these parameters:

```
dspi_master_state_t dspiMasterState; <- the user simply allocates memory for this struct
uint32_t calculatedBaudRate;
dspi_master_user_config_t userConfig; <- the user fills out members for this
    struct
userConfig.isChipSelectContinuous = false;
userConfig.isSckContinuous = false;
userConfig.pcsPolarity = kDspiPcs_ActiveLow;
userConfig.whichCtar = kDspiCtar0;
userConfig.whichPcs = kDspiPcs0;
edma_software_tcd_t stcdSrc2CmdDataLast; <- needs to be aligned to a 32-byte boundary
DSPI_DRV_MasterInit(masterInstance, &dspiMasterState, &userConfig, &stcdSrc2CmdDataLast)
;
```

This initialization function also configures the eDMA module by requesting channels for DMA operation and sets a "use_dma" flag in the run-time state structure to notify transfer functions to use DMA driven operations.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
<i>dspiState</i>	The pointer to the DSPI master driver state structure. The user must pass the memory for this run-time state structure and the DSPI master driver fills out the members. This run-time state structure keeps track of the transfer in progress.
<i>userConfig</i>	The dspi_master_user_config_t user configuration structure. The user must fill out the members of this structure and pass the pointer of this structure into the function.
<i>stcdSrc2CmdDataLast</i>	This is a pointer to a structure of type <code>stcdSrc2CmdDataLast</code> . This pointer needs to be aligned to a 32-byte boundary. Some compilers allow you to use a #pragma directive to align a variable to a desired boundary. The following is an example on how to align this variable to a 32-byte boundary: <pre>#pragma data_alignment=32 <- Normally this is defined as a global variable edma_software_tcd_t stcdSrc2CmdDataLast;</pre>

Returns

An error code or `kStatus_DSPI_Success`.

Note: The pointer parameter "stcdSrc2CmdDataLast" needs to be aligned to a 32-byte boundary. Also note, the only DSPI instance supported for DMA based operation is instance 0.

8.2.2.3 void DSPI_DRV_MasterDeinit (uint32_t *instance*)

This function resets the DSPI peripheral, gates its clock, and disables the interrupt to the core.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
-----------------	---

8.2.2.4 dspi_status_t DSPI_DRV_MasterSetDelay (uint32_t *instance*, dspi_delay_type_t *whichDelay*, uint32_t *delayInNanoSec*, uint32_t * *calculatedDelay*)

This function allows the user to take advantage of the DSPI module's delay options in order to "fine tune" some of the signal timings to match the timing needs of a slower peripheral device. This is an optional function that can be called after the DSPI module has been initialized for master mode. The bus timing delays that can be adjusted are listed below:

PCS to SCK Delay: Adjustable delay option between the assertion of the PCS signal to the first SCK edge.

After SCK Delay: Adjustable delay option between the last edge of SCK to the de-assertion of the PCS signal.

DSPI Master Driver

Delay after Transfer: Adjustable delay option between the de-assertion of the PCS signal for a frame to the assertion of the PCS signal for the next frame. Note this is not adjustable for continuous clock mode as this delay is fixed at one SCK period.

Each of the above delay parameters use both a pre-scalar and scalar value to calculate the needed delay. This function takes in as a parameter the desired delay type and the delay value (in nanoseconds), calculates the values needed for the prescaler and scaler. Returning the actual calculated delay as an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. In addition, the function returns an out-of-range status.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
<i>whichDelay</i>	The desired delay to configure, must be of type <code>dspi_delay_type_t</code>
<i>delayInNano-Sec</i>	The desired delay value in nano-seconds.
<i>calculated-Delay</i>	The calculated delay that best matches the desired delay (in nano-seconds).

Returns

Either `kStatus_DSPI_Success` or `kStatus_DSPI_OutOfRange` if the desired delay exceeds the capability of the device.

8.2.2.5 `dspi_status_t DSPI_DRV_MasterConfigureBus (uint32_t instance, const dsPIC_Device_t * device, uint32_t * calculatedBaudRate)`

The term "device" is used to indicate the SPI device for which the DSPI master is communicating. The user has two options to configure the device parameters: either pass in the pointer to the device configuration structure to the desired transfer function (see `DSPI_DRV_MasterTransferDataBlocking` or `DSPI_DRV_MasterTransferData`) or pass it in to the `DSPI_DRV_MasterConfigureBus` function. The user can pass in a device structure to the transfer function which contains the parameters for the bus (the transfer function then calls this function). However, the user has the option to call this function directly especially to get the calculated baud rate, at which point they may pass in `NULL` for the device structure in the transfer function (assuming they have called this configure bus function first). This is an example to set up the `dsPIC_Device_t` structure to call the `DSPI_DRV_MasterConfigureBus` function by passing in these parameters:

```
dsPIC_Device_t spiDevice;
spiDevice.dataBusConfig.bitsPerFrame = 16;
spiDevice.dataBusConfig.clkPhase = kDspiClockPhase_FirstEdge;
spiDevice.dataBusConfig.clkPolarity = kDspiClockPolarity_ActiveHigh
;
spiDevice.dataBusConfig.direction = kDspiMsbFirst;
spiDevice.bitsPerSec = 50000;
DSPI_DRV_MasterConfigureBus(instance, &spiDevice, &calculatedBaudRate);
```

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
<i>device</i>	Pointer to the device information structure. This structure contains the settings for the SPI bus configuration. The device parameters are the desired baud rate (in bits-per-sec), and the data format field which consists of bits-per-frame, clock polarity and phase, and data shift direction.
<i>calculated-BaudRate</i>	The calculated baud rate passed back to the user to determine if the calculated baud rate is close enough to meet the needs. The baud rate never exceeds the desired baud rate.

Returns

An error code or kStatus_DSPI_Success.

8.2.2.6 **dspi_status_t DSPI_DRV_MasterTransferDataBlocking (uint32_t *instance*, const dspi_device_t *restrict *device*, const uint8_t * *sendBuffer*, uint8_t * *receiveBuffer*, size_t *transferByteCount*, uint32_t *timeout*)**

This function simultaneously sends and receives data on the SPI bus, as SPI is naturally a full-duplex bus. The function does not return until the transfer is complete.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
<i>device</i>	Pointer to the device information structure. This structure contains the settings for the SPI bus configuration in this transfer. You may pass NULL for this parameter, in which case the current bus configuration is used unmodified. The device can be configured separately by calling the DSPI_DRV_MasterConfigureBus function.
<i>sendBuffer</i>	The pointer to the data buffer of the data to send. You may pass NULL for this parameter and bytes with a value of 0 (zero) is sent.
<i>receiveBuffer</i>	Pointer to the buffer where the received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
<i>transferByteCount</i>	The number of bytes to send and receive.

DSPI Master Driver

<i>timeout</i>	A timeout for the transfer in microseconds. If the transfer takes longer than this amount of time, the transfer is aborted and a kStatus_SPI_Timeout error returned.
----------------	--

Return values

kStatus_DSPI_Success	The transfer was successful.
kStatus_DSPI_Busy	Cannot perform transfer because a transfer is already in progress.
kStatus_DSPI_Timeout	The transfer timed out and was aborted.

8.2.2.7 `dspi_status_t DSPI_DRV_MasterTransferData (uint32_t instance, const dspi_device_t *restrict device, const uint8_t * sendBuffer, uint8_t * receiveBuffer, size_t transferByteCount)`

This function returns immediately. The user must check back to check whether the transfer is complete (using the DSPI_DRV_MasterGetTransferStatus function). This function simultaneously sends and receives data on the SPI bus, as SPI is naturally a full-duplex bus.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
<i>device</i>	Pointer to the device information structure. This structure contains the settings for the SPI bus configuration in this transfer. You may pass NULL for this parameter, in which case the current bus configuration is used unmodified. The device can be configured separately by calling the DSPI_DRV_MasterConfigureBus function.
<i>sendBuffer</i>	The pointer to the data buffer of the data to send. You may pass NULL for this parameter, in which case bytes with a value of 0 (zero) are sent.
<i>receiveBuffer</i>	Pointer to the buffer where the received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
<i>transferByteCount</i>	The number of bytes to send and receive.

Return values

kStatus_DSPI_Success	The transfer was successful.
kStatus_DSPI_Busy	Cannot perform transfer because a transfer is already in progress.

8.2.2.8 **dspi_status_t DSPI_DRV_MasterGetTransferStatus (uint32_t *instance*, uint32_t * *framesTransferred*)**

When performing an a-sync transfer, the user can call this function to ascertain the state of the current transfer: in progress (or busy) or complete (success). In addition, if the transfer is still in progress, the user can get the number of words that have been transferred up to now.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
<i>frames-Transferred</i>	Pointer to value that is filled in with the number of frames that have been sent in the active transfer. A frame is defined as the number of bits per frame.

Return values

<i>kStatus_DSPI_Success</i>	The transfer has completed successfully.
<i>kStatus_DSPI_Busy</i>	The transfer is still in progress. <i>framesTransferred</i> is filled with the number of words that have been transferred so far.

8.2.2.9 **dspi_status_t DSPI_DRV_MasterAbortTransfer (uint32_t *instance*)**

During an async transfer, the user has the option to terminate the transfer early if the transfer is still in progress.

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
-----------------	---

Return values

<i>kStatus_DSPI_Success</i>	The transfer was successful.
<i>kStatus_DSPI_No-TransferInProgress</i>	No transfer is currently in progress.

8.3 DSPI Slave Driver

This chapter describes the programming interface of the DSPI slave mode Peripheral driver.

Data Structures

- struct [dspi_slave_callbacks_t](#)
The set of callbacks for the DSPI slave mode. [More...](#)
- struct [dspi_slave_state_t](#)
Runtime state of the DSPI slave driver. [More...](#)
- struct [dspi_slave_user_config_t](#)
User configuration structure and callback functions for the DSPI slave driver. [More...](#)

Initialization and shutdown

- [dspi_status_t DSPI_DRV_SlaveInit](#) (uint32_t instance, [dspi_slave_state_t](#) *dspiState, const [dspi_slave_user_config_t](#) *slaveConfig)
Initializes a DSPI instance for a slave mode operation.
- [dspi_status_t DSPI_DRV_SlaveInitDma](#) (uint32_t instance, [dspi_slave_state_t](#) *dspiState, const [dspi_slave_user_config_t](#) *slaveConfig, const uint8_t *sendBuffer, uint8_t *receiveBuffer, size_t transferByteCount)
Initializes a DSPI instance for slave mode operation with DMA support.
- void [DSPI_DRV_SlaveDeinit](#) (uint32_t instance)
Shuts down a DSPI instance.

8.3.0.10 DSPI Slave Driver

Overview

The DSPI slave peripheral driver supports using the SPI peripheral in slave mode.

Data transfer is performed entirely through callback functions.

Runtime state of the DSPI slave driver

This structure holds data that the DSPI slave peripheral driver uses to communicate between the transfer function and the interrupt handler. The user needs to pass the memory for this structure and the driver fills out the members.

Callbacks

To use this driver in an interrupt driven mode, first define application callbacks. These are the callback functions that are set in the [dspi_slave_callbacks_t](#) structure.

The three callbacks are:

- Data source
- Data sink
- Error notification

The first two callbacks are used to send and receive data. The third callback is invoked if an error occurs.

The prototypes for the three callbacks are:

```
status_t data_source(uint8_t * sourceWord, uint32_t instance);
status_t data_sink(uint8_t sinkWord, uint32_t instance);
void on_error(status_t error, uint32_t instance);
```

All callbacks are invoked from the IRQ state.

Setup

To initialize the DSPI slave driver, first create and fill in a `dsPIC33F_DSPI_SlaveUserConfig_t` structure. This structure defines the callbacks and the data format settings for the SPI peripheral. The structure is not required after the driver is initialized and can be allocated on the stack. The user also must pass the memory for the run-time state structure.

Here is an example of a configuration structure definition:

```
/* Set up and init the slave */
dsPIC33F_DSPI_SlaveUserConfig_t dsPIC33F_DSPI_SlaveUserConfig;
dsPIC33F_DSPI_SlaveUserConfig.callbacks.dataSink = data_sink;
dsPIC33F_DSPI_SlaveUserConfig.callbacks.dataSource = data_source;
dsPIC33F_DSPI_SlaveUserConfig.callbacks.onError = on_error;
dsPIC33F_DSPI_SlaveUserConfig.dataConfig.bitsPerFrame = 16;
dsPIC33F_DSPI_SlaveUserConfig.dataConfig.clkPhase =
    kDsPIC33F_DSPI_ClockPhase_FirstEdge;
dsPIC33F_DSPI_SlaveUserConfig.dataConfig.clkPolarity =
    kDsPIC33F_DSPI_ClockPolarity_ActiveHigh;
dsPIC33F_DSPI_SlaveUserConfig.isModifiedTimingFormatEnabled = false;

DSPI_DRV_SlaveInit(slaveInstance, &dsPIC33F_DSPI_SlaveUserConfig, &dsPIC33F_DSPI_SlaveState);
```

Additionally, the DSPI supports DMA transfers, but only for DSPI instance 0.

The usage of the DSPI slave mode with DMA is slightly different than that of the interrupt driven driver. In the case of DMA usage, the user also passes in the send and receive buffer pointers along with the expected amount of data that wish to transfer. This is needed by the DMA engine.

There is no need to define callbacks for the data source and sink. However, a callback is still needed to handle errors. Also, an additional new callback is needed from the user to handle the completion of the DMA transfer, which can be something as simple as setting a global flag to indicate the transfer is done. An example usage follows.

To use the DSPI with DMA, simply call an alternate initialization function: `DSPI_DRV_SlaveInitDma`:

DSPI Slave Driver

```
/* Function prototype */
dspi_status_t DSPI_DRV_SlaveInitDma(uint32_t instance,
                                      dspi_slave_state_t * dspiState,
                                      const dspi_slave_user_config_t * slaveConfig,
                                      const uint8_t * sendBuffer,
                                      uint8_t * receiveBuffer,
                                      size_t transferByteCount);
```

Example usage:

```
instance = slaveInstance; <- the desired module instance number
dspi_slave_state_t dspiSlaveState; <- the user simply allocates memory for this struct
dspi_slave_user_config_t slaveUserConfig;
slaveUserConfig.callbacks.onError = on_error; <- set to user implementation of function
slaveUserConfig.callbacks.dspiDone = dspi_Dma_Done; <- user defined callback
slaveUserConfig.dataConfig.bitsPerFrame = 16; <- example, can be 4 to 32
slaveUserConfig.dataConfig.clkPhase = kDspiClockPhase_FirstEdge; <- example
    setting
slaveUserConfig.dataConfig.clkPolarity = kDspiClockPolarity_ActiveHigh; <-
    example setting
sendBuffer <- (pointer) to the source data buffer, can be NULL
receiveBuffer <- (pointer) to the receive data buffer, can be NULL
transferCount <- number of bytes to transfer

DSPI_DRV_SlaveInitDma(slaveInstance, &slaveUserConfig, &dspiSlaveState,
                      &sendBuffer, &receiveBuffer, transferCount);

/* Here is an example of the callback to indicate completion of the DMA. Note
 * the name of this callback function matches the name used in the structure member above.
 */
uint32_t g_transferDoneFlag = 0;

static void dspi_Dma_Done(void)
{
    /* set the global flag to indicate transfer done */
    g_transferDoneFlag = 1;
}
```

De-initialization

To de-initialize or shut down the DSPI module, call the function:

```
void DSPI_DRV_SlaveDeinit(slaveInstance);
```

8.3.1 Data Structure Documentation

8.3.1.1 struct dspi_slave_callbacks_t

The user creates the function implementations.

Data Fields

- **dspi_status_t(* dataSource)(uint8_t *sourceWord, uint32_t instance)**
Callback to get word to transmit.

- `dspi_status_t(* dataSink)(uint8_t sinkWord, uint32_t instance)`
Callback to put received word.
- `void(* onError)(dspi_status_t error, uint32_t instance)`
Callback to report a DSPI error, such as an under-run or over-run error.
- `void(* dspiDone)(void)`
Callback to report the slave SPI DMA is done transferring data.

8.3.1.1.0.17 Field Documentation

8.3.1.1.0.17.1 `dspi_status_t(* dspi_slave_callbacks_t::dataSource)(uint8_t *sourceWord, uint32_t instance)`

8.3.1.1.0.17.2 `dspi_status_t(* dspi_slave_callbacks_t::dataSink)(uint8_t sinkWord, uint32_t instance)`

8.3.1.1.0.17.3 `void(* dspi_slave_callbacks_t::onError)(dspi_status_t error, uint32_t instance)`

8.3.1.1.0.17.4 `void(* dspi_slave_callbacks_t::dspiDone)(void)`

Used only for DMA enabled slave SPI operation and not used for interrupt operation.

8.3.1.2 struct dspi_slave_state_t

This structure holds data that is used by the DSPI slave peripheral driver to communicate between the transfer function and the interrupt handler. The user needs to pass in the memory for this structure and the driver fills out the members.

Data Fields

- `dspi_slave_callbacks_t callbacks`
Application/user callbacks.
- `uint32_t bitsPerFrame`
Desired number of bits per frame.
- `bool useDma`
User option to invoke usage of DMA.
- `edma_chn_state_t dmaSrc2TxFifo`
Structure definition for the eDMA channel.
- `edma_chn_state_t dmaRxFifo2RxBuff`
Structure definition for the eDMA channel.

8.3.1.3 struct dspi_slave_user_config_t

Data Fields

- `dspi_slave_callbacks_t callbacks`
Application/user callbacks.
- `dspi_data_format_config_t dataConfig`

DSPI Slave Driver

Data format configuration structure.

8.3.1.3.0.18 Field Documentation

8.3.1.3.0.18.1 `dspi_slave_callbacks_t dspi_slave_user_config_t::callbacks`

8.3.2 Function Documentation

8.3.2.1 `dspi_status_t DSPI_DRV_SlaveInit (uint32_t instance, dspi_slave_state_t * dspiState, const dspi_slave_user_config_t * slaveConfig)`

This function saves the callbacks to the run-time state structure for a later use in the interrupt handler. It also ungates the clock to the DSPI module, initializes the DSPI for slave mode, and enables the module and corresponding interrupts. Once initialized, the DSPI module is configured in slave mode and ready to receive data from the SPI master. This is an example to set up the `dspi_slave_state_t` and the `dspi_slave_user_config_t` parameters and to call the `DSPI_DRV_SlaveInit` function by passing in these parameters:

```
dspi_slave_state_t dspiSlaveState; <- the user simply allocates memory for this structure
dspi_slave_user_config_t slaveUserConfig;
slaveUserConfig.callbacks.dataSink = data_sink; <- set to user implementation of function
slaveUserConfig.callbacks.dataSource = data_source; <- set to user implementation of function
slaveUserConfig.callbacks.onError = on_error; <- set to user implementation of function
slaveUserConfig.dataConfig.bitsPerFrame = 16; <- example setting
slaveUserConfig.dataConfig.clkPhase = kDspiClockPhase_FirstEdge; <- example
    setting
slaveUserConfig.dataConfig.clkPolarity = kDspiClockPolarity_ActiveHigh; <-
    example setting
DSPI_DRV_SlaveInit(slaveInstance, &dspiSlaveState, &slaveUserConfig);
```

Parameters

<code>instance</code>	The instance number of the DSPI peripheral.
<code>dspiState</code>	The pointer to the DSPI slave driver state structure.
<code>slaveConfig</code>	The configuration structure <code>dspi_slave_user_config_t</code> which configures the data bus format and also includes the callbacks.

Returns

An error code or `kStatus_DSPI_Success`.

8.3.2.2 `dspi_status_t DSPI_DRV_SlaveInitDma (uint32_t instance, dspi_slave_state_t * dspiState, const dspi_slave_user_config_t * slaveConfig, const uint8_t * sendBuffer, uint8_t * receiveBuffer, size_t transferByteCount)`

This function is exactly like the `DSPI_DRV_SlaveInit` function but in addition, adds DMA support. This function saves the callbacks to the run-time state structure for later use in the interrupt handler. However,

unlike the CPU driven slave driver, there is no need to define callbacks for the data sink or data source since the user passes in buffers for the send and receive data, and the DMA engine uses those buffers. An onError callback is needed to service potential errors seen during a TX FIFO underflow or RX FIFO overflow. The user also passes in a user defined callback for handling end of transfers (dspiDone). These callbacks are set in the `dspi_slave_callbacks_t` structure which is part of the `dspi_slave_user_config_t` structure. See example below. This function also ungates the clock to the DSPI module, initializes the DSPI for slave mode, enables the module and corresponding interrupts and sets up the DMA channels. Once initialized, the DSPI module is configured in slave mode and ready to receive data from a SPI master. The following is an example of how to set up the `dspi_slave_state_t` and the `dspi_slave_user_config_t` parameters and how to call the DSPI_DRV_SlaveInit function by passing in these parameters:

```
instance = slaveInstance; <- the desired module instance number
dspi_slave_state_t dspiSlaveState; <- the user simply allocates memory for this struct
dspi_slave_user_config_t slaveUserConfig;
slaveUserConfig.callbacks.onError = on_error; <- set to user implementation of function
slaveUserConfig.callbacks.dspiDone = dspi_Dma_Done; <- user defined callback
slaveUserConfig.dataConfig.bitsPerFrame = 16; <- example setting
slaveUserConfig.dataConfig.clkPhase = kDspiClockPhase_FirstEdge; <- example
    setting
slaveUserConfig.dataConfig.clkPolarity = kDspiClockPolarity_ActiveHigh; <-
    example setting
sendBuffer <- (pointer) to the source data buffer, can be NULL
receiveBuffer <- (pointer) to the receive data buffer, can be NULL
transferCount <- number of bytes to transfer

DSPI_DRV_SlaveInitDma(slaveInstance, &dspiSlaveState, &slaveUserConfig,
                      &sendBuffer, &receiveBuffer, transferCount);
```

Parameters

<code>instance</code>	The instance number of the DSPI peripheral.
<code>dspiState</code>	The pointer to the DSPI slave driver state structure.
<code>slaveConfig</code>	The configuration structure <code>dspi_slave_user_config_t</code> which configures the data bus format and also includes the callbacks.
<code>sendBuffer</code>	The pointer to the data buffer of the data to send. You may pass NULL for this parameter, in which case bytes with a value of 0 (zero) are sent.
<code>receiveBuffer</code>	Pointer to the buffer where the received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
<code>transferByte-Count</code>	The expected number of bytes to transfer.

Returns

An error code or `kStatus_DSPI_Success`.

8.3.2.3 `void DSPI_DRV_SlaveDeinit (uint32_t instance)`

Resets the DSPI peripheral, disables the interrupt to the core, and gates its clock.

DSPI Slave Driver

Parameters

<i>instance</i>	The instance number of the DSPI peripheral.
-----------------	---

8.4 DSPI Shared IRQ Driver

This chapter describes the programming interface of the DSPI shared IRQ driver for master and slave Peripheral drivers.

Functions

- void [DSPI_DRV_IRQHandler](#) (uint32_t instance)

The function DSPI_DRV_IRQHandler passes IRQ control to either the master or slave driver.

8.4.1 Function Documentation

8.4.1.1 void [DSPI_DRV_IRQHandler \(uint32_t instance \)](#)

The address of the IRQ handlers are checked to make sure they are non-zero before they are called. If the IRQ handler's address is zero, it means that driver was not present in the link (because the IRQ handlers are marked as weak). This would actually be a program error, because it means the master/slave config for the IRQ was set incorrectly.

Chapter 9

Enhanced Direct Memory Access (eDMA)

The Kinetis SDK provides both HAL and Peripheral drivers for the Direct Memory Access (eDMA) block of Kinetis devices.

Modules

- [DMAMUX HAL driver](#)

This part describes the programming interface of the DMAMUX HAL module.

- [eDMA HAL driver](#)

This part describes the programming interface of the eDMA HAL driver.

- [eDMA Peripheral Driver](#)

This part describes the programming interface of the eDMA Peripheral driver.

- [eDMA request](#)

This part describes the programming interface of the eDMA DMA request resource.

eDMA HAL driver

9.1 eDMA HAL driver

This chapter describes the programming interface of the eDMA HAL driver.

Data Structures

- struct `edma_transfer_config_t`
eDMA transfer size configuration. [More...](#)
- struct `edma_minorloop_offset_config_t`
eDMA TCD Minor loop mapping configuration [More...](#)
- union `edma_error_status_all_t`
Error status of the eDMA module. [More...](#)
- struct `edma_software_tcd_t`
eDMA TCD [More...](#)

Enumerations

- enum `edma_status_t` {
 `kStatus_EDMA_InvalidArgument` = 1U,
 `kStatus_EDMA_Fail` = 2U }
Error code for the eDMA Driver.
- enum `edma_channel_arbitration_t` {
 `kEDMAChnArbitrationFixedPriority` = 0U,
 `kEDMAChnArbitrationRoundrobin` }
eDMA channel arbitration algorithm used for selection among channels.
- enum `edma_channel_priority_t`
eDMA channel priority setting
- enum `edma_modulo_t`
eDMA modulo configuration
- enum `edma_transfer_size_t`
eDMA transfer configuration
- enum `edma_channel_indicator_t` { `kEDMAChannel0` = 0U }
eDMA channel configuration.
- enum `edma_bandwidth_config_t` {
 `kEDMABandwidthStallNone` = 0U,
 `kEDMABandwidthStall4Cycle` = 2U,
 `kEDMABandwidthStall8Cycle` = 3U }
Bandwidth control configuration.

eDMA HAL driver module level operation

- void `EDMA_HAL_Init` (uint32_t baseAddr)
Initializes eDMA module to known state.
- void `EDMA_HAL_CancelTransfer` (uint32_t baseAddr)
Cancels the remaining data transfer.
- void `EDMA_HAL_ErrorCancelTransfer` (uint32_t baseAddr)
Cancels the remaining data transfer and treats it as an error condition.

- static void **EDMA_HAL_SetHaltCmd** (uint32_t baseAddr, bool halt)
Halts/Un-halts the DMA Operations.
- static void **EDMA_HAL_SetHaltOnErrorCmd** (uint32_t baseAddr, bool haltOnError)
Halts or does not halt the eDMA module when an error occurs.
- static void **EDMA_HAL_SetDebugCmd** (uint32_t baseAddr, bool enable)
Enables/Disables the eDMA DEBUG mode.

eDMA HAL driver channel priority and arbitration configuration.

- static void **EDMA_HAL_SetChannelPreemptMode** (uint32_t baseAddr, uint32_t channel, bool preempt, bool preempt)
Sets the preempt and preemption feature for the eDMA channel.
- static void **EDMA_HAL_SetChannelPriority** (uint32_t baseAddr, uint32_t channel, **edma_channel_priority_t** priority)
Sets the eDMA channel priority.
- static void **EDMA_HAL_SetChannelArbitrationMode** (uint32_t baseAddr, **edma_channel_arbitration_t** channelArbitration)
Sets the channel arbitration algorithm.

eDMA HAL driver configuration and operation.

- static void **EDMA_HAL_SetMinorLoopMappingCmd** (uint32_t baseAddr, bool enable)
Enables/Disables the minor loop mapping.
- static void **EDMA_HAL_SetContinuousLinkCmd** (uint32_t baseAddr, bool continuous)
Enables or disables the continuous transfer mode.
- static uint32_t **EDMA_HAL_GetErrorStatus** (uint32_t baseAddr)
Gets the error status of the eDMA module.
- void **EDMA_HAL_SetErrorIntCmd** (uint32_t baseAddr, bool enable, **edma_channel_indicator_t** channel)
Enables/Disables the error interrupt for channels.
- bool **EDMA_HAL_GetErrorIntCmd** (uint32_t baseAddr, uint32_t channel)
Checks whether the eDMA channel error interrupt is enabled or disabled.
- static uint32_t **EDMA_HAL_GetErrorIntStatusFlag** (uint32_t baseAddr)
Gets the eDMA error interrupt status.
- static void **EDMA_HAL_ClearErrorIntStatusFlag** (uint32_t baseAddr, **edma_channel_indicator_t** channel)
Clears the error interrupt status for the eDMA channel or channels.
- void **EDMA_HAL_SetDmaRequestCmd** (uint32_t baseAddr, **edma_channel_indicator_t** channel, bool enable)
Enables/Disables the DMA request for the channel or all channels.
- static bool **EDMA_HAL_GetDmaRequestCmd** (uint32_t baseAddr, uint32_t channel)
Checks whether the eDMA channel DMA request is enabled or disabled.
- static bool **EDMA_HAL_GetDmaRequestStatusFlag** (uint32_t baseAddr, uint32_t channel)
Gets the eDMA channel DMA request status.
- static void **EDMA_HAL_ClearDoneStatusFlag** (uint32_t baseAddr, **edma_channel_indicator_t** channel)
Clears the done status for a channel or all channels.

eDMA HAL driver

- static void **EDMA_HAL_TriggerChannelStart** (uint32_t baseAddr, edma_channel_indicator_t channel)
Triggers the eDMA channel.
- static bool **EDMA_HAL_GetIntStatusFlag** (uint32_t baseAddr, uint32_t channel)
Gets the eDMA channel interrupt request status.
- static uint32_t **EDMA_HAL_GetAllIntStatusFlag** (uint32_t baseAddr)
Gets the eDMA all channel's interrupt request status.
- static void **EDMA_HAL_ClearIntStatusFlag** (uint32_t baseAddr, edma_channel_indicator_t channel)
Clears the interrupt status for the eDMA channel or all channels.

eDMA HAL driver hardware TCD configuration functions.

- void **EDMA_HAL_HTCDClearReg** (uint32_t baseAddr, uint32_t channel)
Clears all registers to 0 for the hardware TCD.
- static void **EDMA_HAL_HTCDSrcAddr** (uint32_t baseAddr, uint32_t channel, uint32_t address)
Configures the source address for the hardware TCD.
- static void **EDMA_HAL_HTCDSrcOffset** (uint32_t baseAddr, uint32_t channel, int16_t offset)
Configures the source address signed offset for the hardware TCD.
- void **EDMA_HAL_HTCDSAttribute** (uint32_t baseAddr, uint32_t channel, edma_modulo_t srcModulo, edma_modulo_t destModulo, edma_transfer_size_t srcTransferSize, edma_transfer_size_t destTransferSize)
Configures the transfer attribute for the eDMA channel.
- void **EDMA_HAL_HTCDSNbytes** (uint32_t baseAddr, uint32_t channel, uint32_t nbytes)
Configures the nbytes for the eDMA channel.
- uint32_t **EDMA_HAL_HTCDGetNbytes** (uint32_t baseAddr, uint32_t channel)
Gets the nbytes configuration data for the hardware TCD.
- void **EDMA_HAL_HTCDSMinorLoopOffset** (uint32_t baseAddr, uint32_t channel, edma_minorloop_offset_config_t *config)
Configures the minor loop offset for the hardware TCD.
- static void **EDMA_HAL_HTCDSrcLastAdjust** (uint32_t baseAddr, uint32_t channel, int32_t size)
Configures the last source address adjustment for the hardware TCD.
- static void **EDMA_HAL_HTCDSDestAddr** (uint32_t baseAddr, uint32_t channel, uint32_t address)
Configures the destination address for the hardware TCD.
- static void **EDMA_HAL_HTCDSDestOffset** (uint32_t baseAddr, uint32_t channel, int16_t offset)
Configures the destination address signed offset for the hardware TCD.
- static void **EDMA_HAL_HTCDSDestLastAdjust** (uint32_t baseAddr, uint32_t channel, uint32_t adjust)
Configures the last source address adjustment.
- void **EDMA_HAL_HTCDSScatterGatherLink** (uint32_t baseAddr, uint32_t channel, edma_software_tcd_t *stcd)
Configures the memory address for the next transfer TCD for the hardware TCD.
- static void **EDMA_HAL_HTCDBandwidth** (uint32_t baseAddr, uint32_t channel, edma_bandwidth_config_t bandwidth)
Configures the bandwidth for the hardware TCD.

- static void [EDMA_HAL_HTCDSethChannelMajorLink](#) (uint32_t baseAddr, uint32_t channel, uint32_t majorChannel, bool enable)

Configures the major channel link for the hardware TCD.
- static uint32_t [EDMA_HAL_HTCDGetMajorLinkChannel](#) (uint32_t baseAddr, uint32_t channel)

Gets the major link channel for the hardware TCD.
- static void [EDMA_HAL_HTCDSethScatterGatherCmd](#) (uint32_t baseAddr, uint32_t channel, bool enable)

Enables/Disables the scatter/gather feature for the hardware TCD.
- static bool [EDMA_HAL_HTCDGetScatterGatherCmd](#) (uint32_t baseAddr, uint32_t channel)

Checks whether the scatter/gather feature is enabled for the hardware TCD.
- static void [EDMA_HAL_HTCDSethDisableDmaRequestAfterTCDDoneCmd](#) (uint32_t baseAddr, uint32_t channel, bool disable)

Disables/Enables the DMA request after the major loop completes for the hardware TCD.
- static void [EDMA_HAL_HTCDSethHalfCompleteIntCmd](#) (uint32_t baseAddr, uint32_t channel, bool enable)

Enables/Disables the half complete interrupt for the hardware TCD.
- static void [EDMA_HAL_HTCDSethIntCmd](#) (uint32_t baseAddr, uint32_t channel, bool enable)

Enables/Disables the interrupt after the major loop completes for the hardware TCD.
- static void [EDMA_HAL_HTCDTriggerChannelStart](#) (uint32_t baseAddr, uint32_t channel)

Triggers the start bits for the hardware TCD.
- static bool [EDMA_HAL_HTCDGetChannelActiveStatus](#) (uint32_t baseAddr, uint32_t channel)

Checks whether the channel is running for the hardware TCD.
- void [EDMA_HAL_HTCDSethChannelMinorLink](#) (uint32_t baseAddr, uint32_t channel, uint32_t linkChannel, bool enable)

Sets the channel minor link for the hardware TCD.
- void [EDMA_HAL_HTCDSethMajorCount](#) (uint32_t baseAddr, uint32_t channel, uint32_t count)

Sets the major iteration count according to minor loop channel link setting.
- uint32_t [EDMA_HAL_HTCDGetBeginMajorCount](#) (uint32_t baseAddr, uint32_t channel)

Gets the number of beginning major counts for the hardware TCD.
- uint32_t [EDMA_HAL_HTCDGetCurrentMajorCount](#) (uint32_t baseAddr, uint32_t channel)

Gets the number of current major counts for the hardware TCD.
- uint32_t [EDMA_HAL_HTCDGetFinishedBytes](#) (uint32_t baseAddr, uint32_t channel)

Gets the number of bytes already transferred for the hardware TCD.
- uint32_t [EDMA_HAL_HTCDGetUnfinishedBytes](#) (uint32_t baseAddr, uint32_t channel)

Gets the number of bytes haven't transferred for the hardware TCD.
- static bool [EDMA_HAL_HTCDGetDoneStatusFlag](#) (uint32_t baseAddr, uint32_t channel)

Gets the channel done status.

EDMA HAL driver software TCD configuration functions.

- static void [EDMA_HAL_STCDSetSrcAddr](#) ([edma_software_tcd_t](#) *stcd, uint32_t address)

Configures the source address for the software TCD.
- static void [EDMA_HAL_STCDSetSrcOffset](#) ([edma_software_tcd_t](#) *stcd, int16_t offset)

Configures the source address signed offset for the software TCD.
- void [EDMA_HAL_STCDSetAttribute](#) ([edma_software_tcd_t](#) *stcd, [edma_modulo_t](#) srcModulo, [edma_modulo_t](#) destModulo, [edma_transfer_size_t](#) srcTransferSize, [edma_transfer_size_t](#) destTransferSize)

Configures the transfer attribute for software TCD.
- void [EDMA_HAL_STCDSetNbytes](#) (uint32_t baseAddr, [edma_software_tcd_t](#) *stcd, uint32_t-

eDMA HAL driver

t nbytes)

Configures the nbytes for software TCD.

- void [EDMA_HAL_STCDSetsMinorLoopOffset](#) (uint32_t baseAddr, edma_software_tcd_t *stcd, edma_minorloop_offset_config_t *config)
Configures the minorloop offset for the software TCD.
- static void [EDMA_HAL_STCDSetsSrcLastAdjust](#) (edma_software_tcd_t *stcd, int32_t size)
Configures the last source address adjustment for the software TCD.
- static void [EDMA_HAL_STCDSetsDestAddr](#) (edma_software_tcd_t *stcd, uint32_t address)
Configures the destination address for the software TCD.
- static void [EDMA_HAL_STCDSetsDestOffset](#) (edma_software_tcd_t *stcd, int16_t offset)
Configures the destination address signed offset for the software TCD.
- static void [EDMA_HAL_STCDSetsDestLastAdjust](#) (edma_software_tcd_t *stcd, uint32_t adjust)
Configures the last source address adjustment.
- void [EDMA_HAL_STCDSetsScatterGatherLink](#) (edma_software_tcd_t *stcd, edma_software_tcd_t *nextStcd)
Configures the memory address for the next transfer TCD for the software TCD.
- static void [EDMA_HAL_STCDSetsBandwidth](#) (edma_software_tcd_t *stcd, edma_bandwidth_config_t bandwidth)
Configures the bandwidth for the software TCD.
- static void [EDMA_HAL_STCDSetsChannelMajorLink](#) (edma_software_tcd_t *stcd, uint32_t majorChannel, bool enable)
Configures the major channel link the software TCD.
- static void [EDMA_HAL_STCDSetsScatterGatherCmd](#) (edma_software_tcd_t *stcd, bool enable)
Enables/Disables the scatter/gather feature for the software TCD.
- static void [EDMA_HAL_STCDSetsDisableDmaRequestAfterTCDDoneCmd](#) (edma_software_tcd_t *stcd, bool disable)
Disables/Enables the DMA request after the major loop completes for the software TCD.
- static void [EDMA_HAL_STCDSetsHalfCompleteIntCmd](#) (edma_software_tcd_t *stcd, bool enable)
Enables/Disables the half complete interrupt for the software TCD.
- static void [EDMA_HAL_STCDSetsIntCmd](#) (edma_software_tcd_t *stcd, bool enable)
Enables/Disables the interrupt after the major loop completes for the software TCD.
- static void [EDMA_HAL_STCDTriggerChannelStart](#) (edma_software_tcd_t *stcd)
Triggers the start bits for the software TCD.
- void [EDMA_HAL_STCDSetsChannelMinorLink](#) (edma_software_tcd_t *stcd, uint32_t linkChannel, bool enable)
Set Channel minor link for software TCD.
- void [EDMA_HAL_STCDSetsMajorCount](#) (edma_software_tcd_t *stcd, uint32_t count)
Sets the major iteration count according to minor loop channel link setting.
- void [EDMA_HAL_PushSTCDToHTCD](#) (uint32_t baseAddr, uint32_t channel, edma_software_tcd_t *stcd)
Copy the software TCD configuration to the hardware TCD.
- edma_status_t [EDMA_HAL_STCDSetsBasicTransfer](#) (uint32_t baseAddr, edma_software_tcd_t *stcd, edma_transfer_config_t *config, bool enableInt, bool disableDmaRequest)
Set the basic transfer for software TCD.

9.1.0.2 eDMA HAL Driver

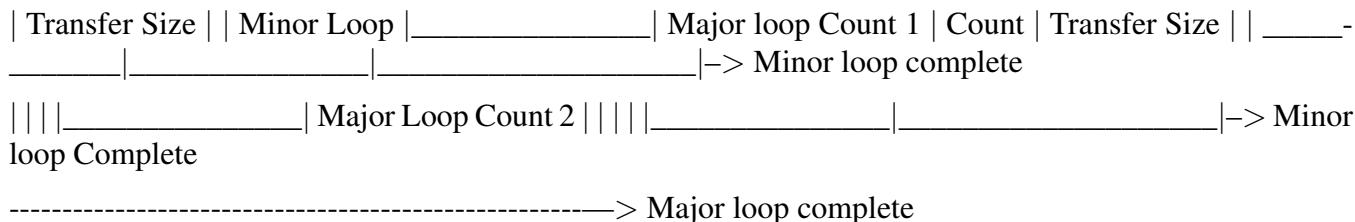
Overview

The eDMA HAL driver provides a function set to operate the eDMA hardware.

9.1.1 Data Structure Documentation

9.1.1.1 struct edma_transfer_config t

This structure configures the basic source/destination transfer attribute. This figure shows the eDMA's transfer model:



Data Fields

- `uint32_t srcAddr`
Memory address pointing to the source data.
 - `uint32_t destAddr`
Memory address pointing to the destination data.
 - `edma_transfer_size_t srcTransferSize`
Source data transfer size.
 - `edma_transfer_size_t destTransferSize`
Destination data transfer size.
 - `int16_t srcOffset`
Sign-extended offset applied to the current source address to form the next-state value as each source read/write is completed.
 - `uint32_t srcLastAddrAdjust`
Last source address adjustment.
 - `uint32_t destLastAddrAdjust`
Last destination address adjustment.
 - `edma_modulo_t srcModulo`
Source address modulo.
 - `edma_modulo_t destModulo`
Destination address modulo.
 - `uint32_t minorLoopCount`
Minor bytes transfer count.
 - `uint16_t majorLoopCount`
Major iteration count.

eDMA HAL driver

9.1.1.1.0.19 Field Documentation

9.1.1.1.0.19.1 `uint32_t edma_transfer_config_t::srcAddr`

9.1.1.1.0.19.2 `uint32_t edma_transfer_config_t::destAddr`

9.1.1.1.0.19.3 `edma_transfer_size_t edma_transfer_config_t::srcTransferSize`

9.1.1.1.0.19.4 `edma_transfer_size_t edma_transfer_config_t::destTransferSize`

9.1.1.1.0.19.5 `int16_t edma_transfer_config_t::srcOffset`

9.1.1.1.0.19.6 `uint32_t edma_transfer_config_t::srcLastAddrAdjust`

9.1.1.1.0.19.7 `uint32_t edma_transfer_config_t::destLastAddrAdjust`

Note here it is only valid when scatter/gather feature is not enabled.

9.1.1.1.0.19.8 `edma_modulo_t edma_transfer_config_t::srcModulo`

9.1.1.1.0.19.9 `edma_modulo_t edma_transfer_config_t::destModulo`

9.1.1.1.0.19.10 `uint32_t edma_transfer_config_t::minorLoopCount`

Number of bytes to be transferred in each service request of the channel.

9.1.1.1.0.19.11 `uint16_t edma_transfer_config_t::majorLoopCount`

9.1.1.2 `struct edma_minorloop_offset_config_t`

Data Fields

- `bool enableSrcMinorloop`
Enable(true) or Disable(false) source minor loop offset.
- `bool enableDestMinorloop`
Enable(true) or Disable(false) destination minor loop offset.
- `uint32_t offset`
Offset for minor loop mapping.

9.1.1.2.0.20 Field Documentation

9.1.1.2.0.20.1 `bool edma_minorloop_offset_config_t::enableSrcMinorloop`

9.1.1.2.0.20.2 `bool edma_minorloop_offset_config_t::enableDestMinorloop`

9.1.1.2.0.20.3 `uint32_t edma_minorloop_offset_config_t::offset`

9.1.1.3 `union edma_error_status_all_t`

9.1.1.4 `struct edma_software_tcd_t`

9.1.2 Enumeration Type Documentation

9.1.2.1 `enum edma_status_t`

Enumerator

kStatus_EDMA_InvalidArgument Parameter is invalid.

kStatus_EDMA_Fail Failed operation.

9.1.2.2 `enum edma_channel_arbitration_t`

Enumerator

kEDMAChnArbitrationFixedPriority Fixed Priority arbitration is used for selection among channels.

kEDMAChnArbitrationRoundrobin Round-Robin arbitration is used for selection among channels.

9.1.2.3 `enum edma_channel_indicator_t`

Enumerator

kEDMACHannel0 Channel 0.

9.1.2.4 `enum edma_bandwidth_config_t`

Enumerator

kEDMABandwidthStallNone No eDMA engine stalls.

kEDMABandwidthStall4Cycle eDMA engine stalls for 4 cycles after each read/write.

kEDMABandwidthStall8Cycle eDMA engine stalls for 8 cycles after each read/write.

9.1.3 Function Documentation

9.1.3.1 `void EDMA_HAL_Init(uint32_t baseAddr)`

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
-----------------	--

9.1.3.2 void EDMA_HAL_CancelTransfer (uint32_t *baseAddr*)

This function stops the executing channel and forces the minor loop to finish. The cancellation takes effect after the last write of the current read/write sequence. The CX clears itself after the cancel has been honored. This cancel retires the channel normally as if the minor loop had completed.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
-----------------	--

9.1.3.3 void EDMA_HAL_ErrorCancelTransfer (uint32_t *baseAddr*)

This function stops the executing channel and forces the minor loop to finish. The cancellation takes effect after the last write of the current read/write sequence. The CX clears itself after the cancel has been honored. This cancel retires the channel normally as if the minor loop had completed. Additional thing is to treat this operation as an error condition.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
-----------------	--

9.1.3.4 static void EDMA_HAL_SetHaltCmd (uint32_t *baseAddr*, bool *halt*) [inline], [static]

This function stalls/un-stalls the start of any new channels. Executing channels are allowed to be completed.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>halt</i>	Halts (true) or un-halts (false) eDMA transfer.

9.1.3.5 static void EDMA_HAL_SetHaltOnErrorCmd (uint32_t *baseAddr*, bool *haltOnError*) [inline], [static]

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

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>haltOnError</i>	Halts (true) or not halt (false) eDMA module when an error occurs.

9.1.3.6 static void EDMA_HAL_SetDebugCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function enables/disables the eDMA Debug mode. When in debug mode, the DMA stalls the start of a new channel. Executing channels are allowed to complete. Channel execution resumes either when the system exits debug mode or when the EDBG bit is cleared.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>enable</i>	Enables (true) or Disable (false) eDMA module debug mode.

9.1.3.7 static void EDMA_HAL_SetChannelPreemptMode (*uint32_t baseAddr, uint32_t channel, bool preempt, bool preemption*) [inline], [static]

This function sets the preempt and preemption features.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>preempt</i>	eDMA channel can't suspend a lower priority channel (true). eDMA channel can suspend a lower priority channel (false).
<i>preemption</i>	eDMA channel can be temporarily suspended by the service request of a higher priority channel (true). eDMA channel can't be suspended by a higher priority channel (false).

9.1.3.8 static void EDMA_HAL_SetChannelPriority (*uint32_t baseAddr, uint32_t channel, edma_channel_priority_t priority*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>priority</i>	Priority of the DMA channel. Different channels should have different priority setting inside a group.

9.1.3.9 static void EDMA_HAL_SetChannelArbitrationMode (uint32_t *baseAddr*, edma_channel_arbitration_t *channelArbitration*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel-Arbitration</i>	Round-Robin way for fixed priority way.

9.1.3.10 static void EDMA_HAL_SetMinorLoopMappingCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables/disables the minor loop mapping feature. If enabled, the NBYTES is redefined to include the individual enable fields and the NBYTES field. The individual enable fields allow the minor loop offset to be applied to the source address, the destination address, or both. The NBYTES field is reduced when either offset is enabled.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>enable</i>	Enables (true) or Disable (false) minor loop mapping.

9.1.3.11 static void EDMA_HAL_SetContinuousLinkCmd (uint32_t *baseAddr*, bool *continuous*) [inline], [static]

This function enables or disables the continuous transfer. If set, a minor loop channel link does not go through the channel arbitration before being activated again. Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself.

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>continuous</i>	Enables (true) or Disable (false) continuous transfer mode.

9.1.3.12 static uint32_t EDMA_HAL_GetErrorStatus (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
-----------------	--

Returns

Detailed information of the error type in the eDMA module.

9.1.3.13 void EDMA_HAL_SetErrorIntCmd (uint32_t *baseAddr*, bool *enable*, edma_channel_indicator_t *channel*)

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>enable</i>	Enable(true) or Disable (false) error interrupt.
<i>channel</i>	Channel indicator. If kEDMAAllChannel is selected, all channels' error interrupt will be enabled/disabled.

9.1.3.14 bool EDMA_HAL_GetErrorIntCmd (uint32_t *baseAddr*, uint32_t *channel*)

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

Error interrupt is enabled (true) or disabled (false).

9.1.3.15 static uint32_t EDMA_HAL_GetErrorIntStatusFlag (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

32 bit variable indicating error channels. If error happens on eDMA channel n, the bit n of this variable is '1'. If not, the bit n of this variable is '0'.

9.1.3.16 static void EDMA_HAL_ClearErrorIntStatusFlag (*uint32_t baseAddr*, *edma_channel_indicator_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>enable</i>	Enable(true) or Disable (false) error interrupt.
<i>channel</i>	Channel indicator. If kEDMAAllChannel is selected, all channels' error interrupt status will be cleared.

9.1.3.17 void EDMA_HAL_SetDmaRequestCmd (*uint32_t baseAddr*, *edma_channel_indicator_t channel*, *bool enable*)

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>enable</i>	Enable(true) or Disable (false) DMA request.
<i>channel</i>	Channel indicator. If kEDMAAllChannel is selected, all channels DMA request are enabled/disabled.

9.1.3.18 static bool EDMA_HAL_GetDmaRequestCmd (*uint32_t baseAddr*, *uint32_t channel*) [inline], [static]

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

DMA request is enabled (true) or disabled (false).

9.1.3.19 static bool EDMA_HAL_GetDmaRequestStatusFlag (*uint32_t baseAddr, uint32_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

Hardware request is triggered in this eDMA channel (true) or not be triggered in this channel (false).

9.1.3.20 static void EDMA_HAL_ClearDoneStatusFlag (*uint32_t baseAddr, edma_channel_indicator_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	Channel indicator. If kEDMAAllChannel is selected, all channels' done status will be cleared.

9.1.3.21 static void EDMA_HAL_TriggerChannelStart (*uint32_t baseAddr, edma_channel_indicator_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	Channel indicator. If kEDMAAllChannel is selected, all channels are triggered.

9.1.3.22 static bool EDMA_HAL_GetIntStatusFlag (uint32_t *baseAddr*, uint32_t *channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

Interrupt request happens in this eDMA channel (true) or not happen in this channel (false).

9.1.3.23 static uint32_t EDMA_HAL_GetAllIntStatusFlag (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

Interrupt status flag of all channels.

9.1.3.24 static void EDMA_HAL_ClearIntStatusFlag (uint32_t *baseAddr*, edma_channel_indicator_t *channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>enable</i>	Enable(true) or Disable (false) error interrupt.
<i>channel</i>	Channel indicator. If kEDMAAllChannel is selected, all channels' interrupt status will be cleared.

9.1.3.25 void EDMA_HAL_HTCDClearReg (uint32_t *baseAddr*, uint32_t *channel*)

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

9.1.3.26 static void EDMA_HAL_HTCDSrcAddr (uint32_t *baseAddr*, uint32_t *channel*, uint32_t *address*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>address</i>	The pointer to the source memory address.

9.1.3.27 static void EDMA_HAL_HTCDSrcOffset (uint32_t *baseAddr*, uint32_t *channel*, int16_t *offset*) [inline], [static]

Sign-extended offset applied to the current source address to form the next-state value as each source read is complete.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>offset</i>	signed-offset for source address.

9.1.3.28 void EDMA_HAL_HTCDSrcAttribute (uint32_t *baseAddr*, uint32_t *channel*, edma_modulo_t *srcModulo*, edma_modulo_t *destModulo*, edma_transfer_size_t *srcTransferSize*, edma_transfer_size_t *destTransferSize*)

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

<i>srcModulo</i>	enumeration type for an allowed source modulo. The value defines a specific address range specified as the value after the SADDR + SOFF calculation is performed on the original register value. Setting this field provides the ability to implement a circular data. For data queues requiring power-of-2 size bytes, the queue should start at a 0-modulo-size address and the SMOD field should be set to the appropriate value for the queue, freezing the desired number of upper address bits. The value programmed into this field specifies the number of the lower address bits allowed to change. For a circular queue application, the SOFF is typically set to the transfer size to implement post-increment addressing with SMOD function restricting the addresses to a 0-modulo-size range.
<i>destModulo</i>	Enum type for an allowed destination modulo.
<i>srcTransferSize</i>	Enum type for source transfer size.
<i>destTransferSize</i>	Enum type for destination transfer size.

9.1.3.29 void EDMA_HAL_HTCDSethNbytes (uint32_t *baseAddr*, uint32_t *channel*, uint32_t *nbytes*)

Note here that user need firstly configure the minor loop mapping feature and then call this function.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>nbytes</i>	Number of bytes to be transferred in each service request of the channel

9.1.3.30 uint32_t EDMA_HAL_HTCDGetNbytes (uint32_t *baseAddr*, uint32_t *channel*)

This function decides whether the minor loop mapping is enabled or whether the source/dest minor loop mapping is enabled. Then, the nbytes are returned accordingly.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

nbytes configuration according to minor loop setting.

eDMA HAL driver

9.1.3.31 void EDMA_HAL_HTCDSetsMinorLoopOffset (uint32_t *baseAddr*, uint32_t *channel*, edma_minorloop_offset_config_t * *config*)

Configures both the enable bits and the offset value. If neither source nor destination offset is enabled, offset is not configured. Note here if source or destination offset is required, the eDMA module EMLM bit will be set in this function. User need to know this side effect.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>config</i>	Configuration data structure for the minor loop offset

9.1.3.32 static void EDMA_HAL_HTCDSetsSrcLastAdjust (uint32_t *baseAddr*, uint32_t *channel*, int32_t *size*) [inline], [static]

Adjustment value added to the source address at the completion of the major iteration count. This value can be applied to restore the source address to the initial value, or adjust the address to reference the next data structure.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>size</i>	adjustment value

9.1.3.33 static void EDMA_HAL_HTCDSetsDestAddr (uint32_t *baseAddr*, uint32_t *channel*, uint32_t *address*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>address</i>	The pointer to the destination address.

9.1.3.34 static void EDMA_HAL_HTCDSetsDestOffset (uint32_t *baseAddr*, uint32_t *channel*, int16_t *offset*) [inline], [static]

Sign-extended offset applied to the current source address to form the next-state value as each destination write is complete.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>offset</i>	signed-offset

9.1.3.35 static void EDMA_HAL_HTCDSetsDestLastAdjust (*uint32_t baseAddr, uint32_t channel, uint32_t adjust*) [inline], [static]

This function adds an adjustment value added to the source address at the completion of the major iteration count. This value can be applied to restore the source address to the initial value, or adjust the address to reference the next data structure.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>adjust</i>	adjustment value

9.1.3.36 void EDMA_HAL_HTCDSetsScatterGatherLink (*uint32_t baseAddr, uint32_t channel, edma_software_tcd_t * stcd*)

This function enables the scatter/gather feature for the hardware TCD and configures the next TCD's address. This address points to the beginning of a 0-modulo-32 byte region containing the next transfer T-CD to be loaded into this channel. The channel reload is performed as the major iteration count completes. The scatter/gather address must be 0-modulo-32-byte. Otherwise, a configuration error is reported.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>stcd</i>	The pointer to the TCD to be linked to this hardware TCD.

9.1.3.37 static void EDMA_HAL_HTCDSetsBandwidth (*uint32_t baseAddr, uint32_t channel, edma_bandwidth_config_t bandwidth*) [inline], [static]

Throttles the amount of bus bandwidth consumed by the eDMA. In general, as the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. This field forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>bandwidth</i>	enum type for bandwidth control

9.1.3.38 static void EDMA_HAL_HTCDSetsChannelMajorLink (*uint32_t baseAddr, uint32_t channel, uint32_t majorChannel, bool enable*) [inline], [static]

If the major link is enabled, after the major loop counter is exhausted, the eDMA engine initiates a channel service request at the channel defined by these six bits by setting that channel start bits.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>majorChannel</i>	channel number for major link
<i>enable</i>	Enables (true) or Disables (false) channel major link.

9.1.3.39 static uint32_t EDMA_HAL_HTCGGetMajorLinkChannel (*uint32_t baseAddr, uint32_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

major link channel number

9.1.3.40 static void EDMA_HAL_HTCDSetsScatterGatherCmd (*uint32_t baseAddr, uint32_t channel, bool enable*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>enable</i>	Enables (true) /Disables (false) scatter/gather feature.

9.1.3.41 static bool EDMA_HAL_HTCDFGetScatterGatherCmd (*uint32_t baseAddr, uint32_t channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

True stand for enabled. False stands for disabled.

9.1.3.42 static void EDMA_HAL_HTCDSSetDisableDmaRequestAfterTCDDoneCmd (*uint32_t baseAddr, uint32_t channel, bool disable*) [inline], [static]

If disabled, the eDMA hardware automatically clears the corresponding DMA request when the current major iteration count reaches zero.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>disable</i>	Disable (true)/Enable (true) DMA request after TCD complete.

9.1.3.43 static void EDMA_HAL_HTCDSHalfCompleteIntCmd (*uint32_t baseAddr, uint32_t channel, bool enable*) [inline], [static]

If set, the channel generates an interrupt request by setting the appropriate bit in the interrupt register when the current major iteration count reaches the halfway point. Specifically, the comparison performed by the eDMA engine is (CITER == (BITER >> 1)). This half-way point interrupt request is provided to support the double-buffered schemes or other types of data movement where the processor needs an early indication of the transfer's process.

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>enable</i>	Enable (true) /Disable (false) half complete interrupt.

9.1.3.44 static void EDMA_HAL_HTCDS.SetIntCmd (uint32_t *baseAddr*, uint32_t *channel*, bool *enable*) [inline], [static]

If enabled, the channel generates an interrupt request by setting the appropriate bit in the interrupt register when the current major iteration count reaches zero.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>enable</i>	Enable (true) /Disable (false) interrupt after TCD done.

9.1.3.45 static void EDMA_HAL_HTCDFTriggerChannelStart (uint32_t *baseAddr*, uint32_t *channel*) [inline], [static]

The eDMA hardware automatically clears this flag after the channel begins execution.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

9.1.3.46 static bool EDMA_HAL_HTCDFGetChannelActiveStatus (uint32_t *baseAddr*, uint32_t *channel*) [inline], [static]

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

True stands for running. False stands for not.

9.1.3.47 void EDMA_HAL_HTCDSethChannelMinorLink (*uint32_t baseAddr, uint32_t channel, uint32_t linkChannel, bool enable*)

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>linkChannel</i>	Channel to be linked on minor loop complete.
<i>enable</i>	Enable (true)/Disable (false) channel minor link.

9.1.3.48 void EDMA_HAL_HTCDSmajorCount (*uint32_t baseAddr, uint32_t channel, uint32_t count*)

Note here that user need to first set the minor loop channel link and then call this function. The execute flow inside this function is dependent on the minor loop channel link setting.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>count</i>	major loop count

9.1.3.49 uint32_t EDMA_HAL_HTCDBeginMajorCount (*uint32_t baseAddr, uint32_t channel*)

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

Begin major counts.

9.1.3.50 uint32_t EDMA_HAL_HTCDCurrentMajorCount (*uint32_t baseAddr, uint32_t channel*)

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

Current major counts.

9.1.3.51 `uint32_t EDMA_HAL_HTCDFinishedBytes (uint32_t baseAddr, uint32_t channel)`

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

data bytes already transferred

9.1.3.52 `uint32_t EDMA_HAL_HTCDFinishedBytes (uint32_t baseAddr, uint32_t channel)`

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

data bytes already transferred

9.1.3.53 `static bool EDMA_HAL_HTCDDoneStatusFlag (uint32_t baseAddr, uint32_t channel) [inline], [static]`

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.

Returns

If channel done.

9.1.3.54 static void EDMA_HAL_STCDSetSrcAddr (*edma_software_tcd_t * stcd*, *uint32_t address*) [inline], [static]

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>channel</i>	eDMA channel number.
<i>address</i>	The pointer to the source memory address.

9.1.3.55 static void EDMA_HAL_STCDSetSrcOffset (*edma_software_tcd_t * stcd*, *int16_t offset*) [inline], [static]

Sign-extended offset applied to the current source address to form the next-state value as each source read is complete.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>offset</i>	signed-offset for source address.

**9.1.3.56 void EDMA_HAL_STCDSetAttribute (edma_software_tcd_t * *stcd*,
edma_modulo_t *srcModulo*, edma_modulo_t *destModulo*, edma_transfer_size_t
srcTransferSize, edma_transfer_size_t *destTransferSize*)**

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>srcModulo</i>	enum type for an allowed source modulo. The value defines a specific address range specified as the value after the SADDR + SOFF calculation is performed on the original register value. Setting this field provides the ability to implement a circular data. For data queues requiring power-of-2 size bytes, the queue should start at a 0-modulo-size address and the SMOD field should be set to the appropriate value for the queue, freezing the desired number of upper address bits. The value programmed into this field specifies the number of the lower address bits allowed to change. For a circular queue application, the SOFF is typically set to the transfer size to implement post-increment addressing with SMOD function restricting the addresses to a 0-modulo-size range.
<i>destModulo</i>	Enum type for an allowed destination modulo.
<i>srcTransferSize</i>	Enum type for source transfer size.
<i>destTransferSize</i>	Enum type for destination transfer size.

9.1.3.57 void EDMA_HAL_STCDSetNbytes (uint32_t *baseAddr*, edma_software_tcd_t * *stcd*, uint32_t *nbytes*)

Note here that user need firstly configure the minor loop mapping feature and then call this function.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>stcd</i>	The pointer to the software TCD.
<i>nbytes</i>	Number of bytes to be transferred in each service request of the channel

**9.1.3.58 void EDMA_HAL_STCDSetMinorLoopOffset (uint32_t *baseAddr*,
edma_software_tcd_t * *stcd*, edma_minorloop_offset_config_t * *config*)**

Configures both the enable bits and the offset value. If neither source nor dest offset is enabled, offset is not configured. Note here if source or destination offset is required, the eDMA module EMLM bit will be set in this function. User need to know this side effect.

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>stcd</i>	The pointer to the software TCD.
<i>config</i>	Configuration data structure for the minorloop offset

9.1.3.59 static void EDMA_HAL_STCDSetSrcLastAdjust (*edma_software_tcd_t * stcd*, *int32_t size*) [inline], [static]

Adjustment value added to the source address at the completion of the major iteration count. This value can be applied to restore the source address to the initial value, or adjust the address to reference the next data structure.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>size</i>	adjustment value

9.1.3.60 static void EDMA_HAL_STCDSetDestAddr (*edma_software_tcd_t * stcd*, *uint32_t address*) [inline], [static]

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>address</i>	The pointer to the destination addresss.

9.1.3.61 static void EDMA_HAL_STCDSetDestOffset (*edma_software_tcd_t * stcd*, *int16_t offset*) [inline], [static]

Sign-extended offset applied to the current source address to form the next-state value as each destination write is complete.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>offset</i>	signed-offset

9.1.3.62 static void EDMA_HAL_STCDSetDestLastAdjust (*edma_software_tcd_t * stcd*, *uint32_t adjust*) [inline], [static]

This function add an adjustment value added to the source address at the completion of the major iteration count. This value can be applied to restore the source address to the initial value, or adjust the address to reference the next data structure.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>adjust</i>	adjustment value

9.1.3.63 void EDMA_HAL_STCDSetScatterGatherLink (*edma_software_tcd_t * stcd*, *edma_software_tcd_t * nextStcd*)

This function enable the scatter/gather feature for the software TCD and configure the next TCD's address. This address points to the beginning of a 0-modulo-32 byte region containing the next transfer TCD to be loaded into this channel. The channel reload is performed as the major iteration count completes. The scatter/gather address must be 0-modulo-32-byte. Otherwise, a configuration error is reported.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>nextStcd</i>	The pointer to the TCD to be linked to this software TCD.

9.1.3.64 static void EDMA_HAL_STCDSetBandwidth (*edma_software_tcd_t * stcd*, *edma_bandwidth_config_t bandwidth*) [inline], [static]

Throttles the amount of bus bandwidth consumed by the eDMA. In general, as the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. This field forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>bandwidth</i>	enum type for bandwidth control

eDMA HAL driver

**9.1.3.65 static void EDMA_HAL_STCDSetChannelMajorLink (`edma_software_tcd_t` *
`stcd`, `uint32_t majorChannel`, `bool enable`) [inline], [static]**

If the majorlink is enabled, after the major loop counter is exhausted, the eDMA engine initiates a channel service request at the channel defined by these six bits by setting that channel start bits.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>majorChannel</i>	channel number for major link
<i>enable</i>	Enables (true) or Disables (false) channel major link.

9.1.3.66 static void EDMA_HAL_STCDSetScatterGatherCmd (*edma_software_tcd_t* * *stcd*, *bool enable*) [inline], [static]

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>enable</i>	Enables (true) /Disables (false) scatter/gather feature.

9.1.3.67 static void EDMA_HAL_STCDSetDisableDmaRequestAfterTCDDoneCmd (*edma_software_tcd_t* * *stcd*, *bool disable*) [inline], [static]

If disabled, the eDMA hardware automatically clears the corresponding DMA request when the current major iteration count reaches zero.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>disable</i>	Disable (true)/Enable (true) dma request after TCD complete.

9.1.3.68 static void EDMA_HAL_STCDSetHalfCompleteIntCmd (*edma_software_tcd_t* * *stcd*, *bool enable*) [inline], [static]

If set, the channel generates an interrupt request by setting the appropriate bit in the interrupt register when the current major iteration count reaches the halfway point. Specifically, the comparison performed by the eDMA engine is (CITER == (BITER >> 1)). This half-way point interrupt request is provided to support the double-buffered schemes or other types of data movement where the processor needs an early indication of the transfer's process.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>enable</i>	Enable (true) /Disable (false) half complete interrupt.

eDMA HAL driver

9.1.3.69 static void EDMA_HAL_STCDSetIntCmd (edma_software_tcd_t * *stcd*, bool *enable*) [inline], [static]

If enabled, the channel generates an interrupt request by setting the appropriate bit in the interrupt register when the current major iteration count reaches zero.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>enable</i>	Enable (true) /Disable (false) interrupt after TCD done.

9.1.3.70 static void EDMA_HAL_STCDTriggerChannelStart (edma_software_tcd_t * *stcd*) [inline], [static]

The eDMA hardware automatically clears this flag after the channel begins execution.

Parameters

<i>stcd</i>	The pointer to the software TCD.
-------------	----------------------------------

9.1.3.71 void EDMA_HAL_STCDSetChannelMinorLink (edma_software_tcd_t * *stcd*, uint32_t *linkChannel*, bool *enable*)

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>linkChannel</i>	Channel to be linked on minor loop complete.
<i>enable</i>	Enable (true)/Disable (false) channel minor link.

9.1.3.72 void EDMA_HAL_STCDSetMajorCount (edma_software_tcd_t * *stcd*, uint32_t *count*)

Note here that user need to first set the minor loop channel link and then call this function. The execute flow inside this function is dependent on the minor loop channel link setting.

Parameters

<i>stcd</i>	The pointer to the software TCD.
<i>count</i>	major loop count

9.1.3.73 void EDMA_HAL_PushSTCDToHTCD (uint32_t *baseAddr*, uint32_t *channel*, edma_software_tcd_t * *stcd*)

eDMA HAL driver

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>channel</i>	eDMA channel number.
<i>stcd</i>	The pointer to the software TCD.

9.1.3.74 `edma_status_t EDMA_HAL_STCDSetsBasicTransfer (uint32_t baseAddr, edma_software_tcd_t * stcd, edma_transfer_config_t * config, bool enableInt, bool disableDmaRequest)`

This function is used to setup the basic transfer for software TCD. The minor loop setting is not involved here cause minor loop's configuration will lay a impact on the global eDMA setting. And the source minor loop offset is relevant to the dest minor loop offset. For these reasons, minor loop offset configuration is treated as an advanced configuration. User can call the [EDMA_HAL_STCDSetsMinorLoopOffset\(\)](#) to configure the minor loop offset feature.

Parameters

<i>baseAddr</i>	Register base address for eDMA module.
<i>stcd</i>	The pointer to the software TCD.
<i>config</i>	The pointer to the transfer configuration structure.
<i>enableInt</i>	Enables (true) or Disables (false) interrupt on TCD complete.
<i>disableDmaRequest</i>	Disables (true) or Enable (false) dma request on TCD complete.

9.2 eDMA Peripheral Driver

This chapter describes the programming interface of the eDMA Peripheral driver.

Data Structures

- struct [edma_user_config_t](#)
The user configuration structure for the eDMA driver. [More...](#)
- struct [edma_chn_state_t](#)
Data structure for the eDMA channel. [More...](#)
- struct [edma_scatter_gather_list_t](#)
Data structure for configuring a discrete memory transfer. [More...](#)
- struct [edma_state_t](#)
Runtime state structure for the eDMA driver. [More...](#)

Macros

- #define [STCD_SIZE](#)(number) ((number + 1) * 32)
Macro for size of memory need for software TCD.
- #define [VIRTUAL_CHN_TO_EDMA_MODULE_REGBASE](#)(chn) g_edmaBaseAddr[chn/FSL_FEATURE_EDMA_MODULE_CHANNEL]
Macro to get the eDMA physical module indicator from virtual channel indicator.
- #define [VIRTUAL_CHN_TO_EDMA_CHN](#)(chn) (chn%FSL_FEATURE_EDMA_MODULE_CHANNEL)
Macro to get the eDMA physical channel indicator from virtual channel indicator.
- #define [VIRTUAL_CHN_TO_DMAMUX_MODULE_REGBASE](#)(chn) g_dmamuxBaseAddr[chn/FSL_FEATURE_DMAMUX_MODULE_CHANNEL]
Macro to get the DMAMUX physical module indicator from virtual channel indicator.
- #define [VIRTUAL_CHN_TO_DMAMUX_CHN](#)(chn) (chn%FSL_FEATURE_DMAMUX_MODULE_CHANNEL)
Macro to get the DMAMUX physical channel indicator from virtual channel indicator.

Typedefs

- typedef void(* [edma_callback_t](#))(void *parameter, [edma_chn_status_t](#) status)
Definition for the eDMA channel callback function.

Enumerations

- enum [edma_chn_status_t](#) {

 [kEDMAChnNormal](#) = 0U,

 [kEDMAChnIdle](#),

 [kEDMAChnError](#) }
- Channel status for eDMA channel.*

eDMA Peripheral Driver

- enum `edma_chn_state_type_t` {
 `kEDMAInvalidChannel` = 0xFFU,
 `kEDMAAnyChannel` = 0xFEU }
 enum type for channel allocation.
- enum `edma_transfer_type_t` {
 `kEDMAPeripheralToMemory`,
 `kEDMAMemoryToPeripheral`,
 `kEDMAMemoryToMemory` }
 A type for the DMA transfer.

eDMA peripheral driver module level functions

- `edma_status_t EDMA_DRV_Init` (`edma_state_t` *`edmaState`, const `edma_user_config_t` *`userConfig`)
 Initializes all eDMA modules in an SOC.
- `edma_status_t EDMA_DRV_Deinit` (void)
 Shuts down all eDMA modules.

eDMA peripheral driver channel management functions

- `uint8_t EDMA_DRV_RequestChannel` (`uint8_t` `channel`, `dma_request_source_t` `source`, `edma_chn_state_t` *`chn`)
 Requests an eDMA channel dynamically or statically.
- `edma_status_t EDMA_DRV_ReleaseChannel` (`edma_chn_state_t` *`chn`)
 Releases an eDMA channel.

eDMA peripheral driver transfer setup functions

- static `edma_status_t EDMA_DRV_PrepDescriptorTransfer` (`edma_chn_state_t` *`chn`, `edma_software_tcd_t` *`stcd`, `edma_transfer_config_t` *`config`, bool `enableInt`, bool `disableDmaRequest`)
 Sets the descriptor basic transfer for the descriptor.
- static `edma_status_t EDMA_DRV_PrepDescriptorScatterGather` (`edma_software_tcd_t` *`stcd`, `edma_software_tcd_t` *`nextStcd`)
 Configures the memory address for the next transfer TCD for the software TCD.
- static `edma_status_t EDMA_DRV_PrepDescriptorChannelLink` (`edma_software_tcd_t` *`stcd`, `uint32_t` `linkChn`)
 Configures the major channel link the software TCD.
- `edma_status_t EDMA_DRV_PushDescriptorToReg` (`edma_chn_state_t` *`chn`, `edma_software_tcd_t` *`stcd`)
 Copies the software TCD configuration to the hardware TCD.
- `edma_status_t EDMA_DRV_ConfigLoopTransfer` (`edma_chn_state_t` *`chn`, `edma_software_tcd_t` *`stcd`, `edma_transfer_type_t` `type`, `uint32_t` `srcAddr`, `uint32_t` `destAddr`, `uint32_t` `size`, `uint32_t` `bytesOnEachRequest`, `uint32_t` `totalLength`, `uint8_t` `number`)
 Configures the DMA transfer in a scatter-gather mode.

- `edma_status_t EDMA_DRV_ConfigScatterGatherTransfer (edma_chn_state_t *chn, edma_software_tcd_t *stcd, edma_transfer_type_t type, uint32_t size, uint32_t bytesOnEachRequest, edma_scatter_gather_list_t *srcList, edma_scatter_gather_list_t *destList, uint8_t number)`
Configures the DMA transfer in a scatter-gather mode.

eDMA peripheral driver channel operation functions

- `edma_status_t EDMA_DRV_StartChannel (edma_chn_state_t *chn)`
Starts an eDMA channel.
- `edma_status_t EDMA_DRV_StopChannel (edma_chn_state_t *chn)`
Stops the eDMA channel.

eDMA peripheral callback and interrupt functions

- `edma_status_t EDMA_DRV_InstallCallback (edma_chn_state_t *chn, edma_callback_t callback, void *parameter)`
Registers the callback function and the parameter for eDMA channel.
- `void EDMA_DRV_IRQHandler (uint8_t channel)`
IRQ Handler for eDMA channel interrupt.
- `void EDMA_DRV_ErrorIRQHandler (uint8_t instance)`
ERROR IRQ Handler for eDMA channel interrupt.

eDMA peripheral driver misc functions

- static `edma_chn_status_t EDMA_DRV_GetChannelStatus (edma_chn_state_t *chn)`
Gets the eDMA channel status.
- static `uint32_t EDMA_DRV_GetFinishedBytes (edma_chn_state_t *chn)`
Gets the bytes already transferred for eDMA channel current TCD.

9.2.0.75 eDMA Peripheral Driver

Overview

The eDMA driver requests, configures, and uses eDMA hardware. It supports module initializations and DMA channel configurations.

Initialization

To initialize the DMA module, call the `EDMA_DRV_Init()` function. The user does not need to pass a configuration data structure. This function enables the eDMA module and clock automatically.

eDMA Peripheral Driver

Channel concept

The eDMA module has many channels. Additionally, the EDMA peripheral driver is designed based on a channel concept. All operations should be started by requesting an eDMA channel and ended by freeing an eDMA channel. The user can configure and run operations on the eDMA module by allocating a channel. If a channel is not allocated, a system error may occur.

DMA request concept

A DMA request is used to trigger an eDMA transfer. The DMA request table is available in chip configuration chapters in each chip reference manual.

Memory allocation and alignment

The eDMA peripheral driver does not allocate any memory dynamically. The user needs to provide the allocated memory pointer for the driver and ensure that the memory is valid. If this is not done, a system error may occur. The user needs to prepare three types of memory:

1. The handler memory: [edma_channel_t]. The driver must store the status data for each channel and the edma_channel_t is designed for this purpose.
2. The [edma_software_tcd_t](#). The eDMA supports a TCD chain, which provides either the scatter-gather feature or the loop feature. The eDMA module loads the TCDCs from memory, where TCDs are stored. The user must provide the memory storing software TCDs and the [EDMA_DRV_ConfigScatterGatherTransfer\(\)](#) function or the [EDMA_DRV_ConfigLoopTransfer\(\)](#) function to configure the software TCDs. If those functions fail to configure the software TCDs, use the [EDMA_DRV_PrepDescriptorTransfer\(\)](#) function to configure the TCD. Then, call the [EDMA_DRV_PushDescriptorToReg\(\)](#) function to push the TCD to registers.
3. The status memory. If the user wants to know the status of TCD chains, the user needs to provide the status memory and the driver will fill it with the chain status.
The start address of the software TCDs must be 32 bytes.

Call diagram

To use the DMA driver, follow these steps:

1. Initialize the DMA module: [EDMA_DRV_Init\(\)](#).
2. Request a DMA channel: [EDMA_DRV_RequestChannel\(\)](#).
3. Configure the TCD:
 - Configure the TCD chain in a scatter-gather list. UART transmit/receive is the common case.
 - Configure the TCD chain in a loop way. Audio playback/Record is the common case.
 - Configure software TCD and push it to registers. Use the DSPI case to configure and push the TCD to registers.
4. Register callback function: [EDMA_DRV_InstallCallback](#).

5. Start the DMA channel: EDMA_DRV_StartChannel.
6. [OPTION] Stop the DMA channel: EDMA_DRV_StopChannel.
7. Free the DMA channel: EDMA_DRV_ReleaseChannel.

This is an example code to initialize and configure the driver by configuring the descriptor:

```

EDMA_DRV_Init();

stcd = (edma_software_tcd_t *)(((uint32_t)status + 32) & ~0x1F);

/* Prepare the memory space. */
for ( i = 0; i < kEdmaTestChainLength; i++)
{
    /* Allocate the memory! */
    srcAddr[i] = malloc(kEdmaTestBufferSize);
    destAddr[i] = malloc(kEdmaTestBufferSize);

    /* Check whether the allocation is successfully. */
    if (((uint32_t)srcAddr[i] == 0x0U) & ((uint32_t)destAddr[i] == 0x0U))
    {
        printf("Fail to allocate memory for EDMA test! \r\n");
        goto error;
    }

    /* Init the memory buffer. */
    for (j = 0; j < kEdmaTestBufferSize; j++)
    {
        srcAddr[i][j] = j;
        destAddr[i][j] = 0;
    }

    srcSG[i].address = (uint32_t)srcAddr[i];
    destSG[i].address = (uint32_t)destAddr[i];
    srcSG[i].length = kEdmaTestBufferSize;
    destSG[i].length = kEdmaTestBufferSize;

}

if (EDMA_DRV_RequestChannel(channel, kDmaRequestMux0AlwaysOn62, &chan_handler) !=
    channel)
{
    printf("Failed to request channel %d !\r\n", channel);
    goto error;
}

EDMA_DRV_ConfigScatterGatherTransfer(
    stcd, &chan_handler, kDmaMemoryToMemory,
    0x1U, kEdmaTestWatermarkLevel,
    srcSG, destSG,
    kEdmaTestChainLength);

EDMA_DRV_InstallCallback(&chan_handler, test_callback, &chan_handler);

EDMA_DRV_StartChannel(&chan_handler);

```

For an example to configure the loop mode, see the SAI module driver.

Extend the driver

The user can call the eDMA HAL driver to extend the application capability.

eDMA Peripheral Driver

9.2.1 Data Structure Documentation

9.2.1.1 struct edma_user_config_t

Use an instance of this structure with the [EDMA_DRV_Init\(\)](#) function. This allows the user to configure settings of the EDMA peripheral with a single function call.

Data Fields

- [edma_channel_arbitration_t chnArbitration](#)
eDMA channel arbitration.

9.2.1.0.21 Field Documentation

9.2.1.1.0.21.1 [edma_channel_arbitration_t edma_user_config_t::chnArbitration](#)

9.2.1.2 struct edma_chn_state_t

Data Fields

- [uint8_t channel](#)
Virtual channel indicator.
- [edma_callback_t callback](#)
Callback function pointer for the eDMA channel.
- [void * parameter](#)
Parameter for the callback function pointer.
- [volatile edma_chn_status_t status](#)
eDMA channel status.

9.2.1.2.0.22 Field Documentation

9.2.1.2.0.22.1 [uint8_t edma_chn_state_t::channel](#)

9.2.1.2.0.22.2 [edma_callback_t edma_chn_state_t::callback](#)

It will be called at the eDMA channel complete and eDMA channel error.

9.2.1.2.0.22.3 [void* edma_chn_state_t::parameter](#)

9.2.1.2.0.22.4 [volatile edma_chn_status_t edma_chn_state_t::status](#)

9.2.1.3 struct edma_scatter_gather_list_t

Data Fields

- [uint32_t address](#)
Address of buffer.
- [uint32_t length](#)
Length of buffer.

9.2.1.3.0.23 Field Documentation

9.2.1.3.0.23.1 `uint32_t edma_scatter_gather_list_t::address`

9.2.1.3.0.23.2 `uint32_t edma_scatter_gather_list_t::length`

9.2.1.4 struct `edma_state_t`

This structure holds data that is used by the eDMA peripheral driver to manage multi eDMA channels. The user must pass the memory for this run-time state structure and the eDMA driver fills out the members.

Data Fields

- `edma_chn_state_t *volatile chn [FSL_FEATURE_EDMA_DMAMUX_CHANNELS]`
Pointer array storing channel state.

9.2.1.4.0.24 Field Documentation

9.2.1.4.0.24.1 `edma_chn_state_t* volatile edma_state_t::chn[FSL_FEATURE_EDMA_DMAMUX_CHANNELS]`

9.2.2 Macro Definition Documentation

9.2.2.1 `#define STCD_SIZE(number) ((number + 1) * 32)`

Software TCD is aligned to 32 bytes. To make sure the software TCD can meet the eDMA module's requirement, allocate memory with extra 32 bytes.

9.2.2.2 `#define VIRTUAL_CHN_TO_EDMA_MODULE_REGBASE(chn) g_edmaBaseAddr[chn/FSL_FEATURE_EDMA_MODULE_CHANNEL]`

9.2.2.3 `#define VIRTUAL_CHN_TO_EDMA_CHN(chn) (chn%FSL_FEATURE_EDMA_MODULE_CHANNEL)`

9.2.2.4 `#define VIRTUAL_CHN_TO_DMAMUX_MODULE_REGBASE(chn) g_dmamuxBaseAddr[chn/FSL_FEATURE_DMAMUX_MODULE_CHANNEL]`

9.2.2.5 `#define VIRTUAL_CHN_TO_DMAMUX_CHN(chn) (chn%FSL_FEATURE_DMAMUX_MODULE_CHANNEL)`

9.2.3 Typedef Documentation

9.2.3.1 `typedef void(* edma_callback_t)(void *parameter, edma_chn_status_t status)`

Prototype for the callback function registered in the eDMA driver.

9.2.4 Enumeration Type Documentation

9.2.4.1 enum edma_chn_status_t

A structure describing the eDMA channel status. The user can get the status by callback parameter or by calling EDMA_DRV_getStatus().

Enumerator

kEDMACHnNormal eDMA channel is occupied.

kEDMACHnIdle eDMA channel is idle.

kEDMACHnError An error occurs in the eDMA channel.

9.2.4.2 enum edma_chn_state_type_t

Enumerator

kEDMAInvalidChannel Macros indicate the failure of the channel request.

kEDMAAnyChannel Macros used when requesting channel dynamically.

9.2.4.3 enum edma_transfer_type_t

Enumerator

kEDMAPeripheralToMemory Transfer from peripheral to memory.

kEDMAMemoryToPeripheral Transfer from memory to peripheral.

kEDMAMemoryToMemory Transfer from memory to memory.

9.2.5 Function Documentation

9.2.5.1 edma_status_t EDMA_DRV_Init (edma_state_t * ***edmaState***, const edma_user_config_t * ***userConfig***)

This function initializes the run-time state structure to provide the eDMA channel allocation release, protect, and track the state for channels. This function also opens the clock to the eDMA modules, resets the eDMA modules, initializes the module to user-defined settings and default settings. This is an example to set up the ***edma_state_t*** and the ***edma_user_config_t*** parameters and to call the EDMA_DRV_Init function by passing in these parameters.

```
edma_state_t state;      <- The user simply allocates memory for this structure.  
edma_user_config_t userConfig;    <- The user fills out members for this structure.  
  
userConfig.chnArbitration = kEDMACHnArbitrationRoundrobin;  
#if (FSL_FEATURE_EDMA_CHANNEL_GROUP_COUNT > 0x1U)  
 //configuration for 2 lines below only valid for SoCs with more than one group.
```

```

userConfig.groupArbitration = kEDMAGroupArbitrationFixedPriority;
userConfig.groupPriority = kEDMAGroup0PriorityHighGroup1PriorityLow;
#endif
userConfig.notHaltOnError = false;      <- The default setting is false, means eDMA halt on error.

EDMA_DRV_Init(&state, &userConfig);

```

Parameters

<i>edmaState</i>	The pointer to the eDMA peripheral driver state structure. The user must pass the memory for this run-time state structure and the eDMA peripheral driver will fill out the members. This run-time state structure keeps track of the eDMA channels status. The memory must be kept valid before calling the EDMA_DRV_DeInit.
<i>userConfig</i>	User configuration structure for eDMA peripheral drivers. The user must fill out the members of this structure and pass the pointer of this structure into the function.

Returns

An eDMA error codes or kStatus_EDMA_Success.

9.2.5.2 **edma_status_t EDMA_DRV_Deinit(void)**

This function resets the eDMA modules to reset state, gates the clock, and disables the interrupt to the core.

Returns

An eDMA error codes or kStatus_EDMA_Success.

9.2.5.3 **uint8_t EDMA_DRV_RequestChannel(uint8_t channel, dma_request_source_t source, edma_chn_state_t * chn)**

This function allocates eDMA channel according to the required channel allocation and corresponding to the eDMA hardware request, initializes the channel state memory provided by user and fills out the members. This functions also sets up the hardware request configuration according to the user.

For Kinetis SOC, a hardware request can be mapped to eDMA channels and used for the channel trigger. Some hardware requests can only be mapped to a limited channels. For example, the Kinetis K70FN1M0-VMJ15 SOC eDMA module has 2 eDMA channel groups. The first group consists of the channel 0 - 15. The second group consists of channel 16 - 31. The hardware request UART0-Receive can be only mapped to group 1. Therefore, the hardware request is one of the parameter that the user needs to provide for the channel request. Channel needn't be triggered by the peripheral hardware request. The user can provide the ALWAYSON type hardware request to trigger the channel continuously.

This function provides two ways to allocate an eDMA channel: statically and dynamically. In a static allocation, the user provides the required channel number and eDMA driver tries to allocate the required

eDMA Peripheral Driver

channel to the user. If the channel is not occupied, the eDMA driver is successfully assigned to the user. If the channel is already occupied, the user gets the return value kEDMAInvalidChn. This is an example to request a channel in a static way:

```
uint32_t channelNumber = 14;    <- Try to allocate the channel 14
edma_chn_state_t chn;        <- The user simply allocates memory for this structure.

if ( kEDMAInvalidChannel == EDMA_DRV_RequestChannel(channel,
    kDmaRequestMux0AlwaysOn54, chn))
{
    printf("request channel %d failed!\n", channel);
}
```

In a dynamic allocation, any of the free eDMA channels are available for use. eDMA driver assigns the first free channel to the user. This is an example for user to request a channel dynamically :

```
uint32_t channel;      <- Store the allocated channel number.
edma_chn_state_t chn;    <- The user simply allocates memory for this structure.

channel = EDMA_DRV_RequestChannel(kEDMAAnyChannel,
    kDmaRequestMux0AlwaysOn54, chn);

if (channel == kEDMAInvalidChannel)
{
    printf("request channel %d failed!\n", channel);
}
else
{
    printf("Channel %d is successfully allocated! /n", channel);
}
```

Parameters

<i>channel</i>	Requested channel number. If the chn is assigned with the kEDMAAnyChannel, the eDMA driver allocates the channel dynamically. If the chn is assigned with a valid channel number, the eDMA driver allocates that channel.
<i>source</i>	eDMA hardware request number.
<i>chn</i>	The pointer to the eDMA channel state structure. The user must pass the memory for this run-time state structure and the eDMA peripheral driver fills out the members. This run-time state structure keeps tracks of the eDMA channel status. The memory must be kept valid before calling the EDMA_DRV_ReleaseChannel() .

Returns

Successfully allocated channel number or the kEDMAInvalidChannel indicating that the request is failed.

9.2.5.4 **edma_status_t EDMA_DRV_ReleaseChannel (edma_chn_state_t * chn)**

This function stops the eDMA channel and disables the interrupt of this channel. The channel state structure can be released after this function is called.

Parameters

<i>chn</i>	The pointer to the channel state structure.
------------	---

Returns

An eDMA error codes or kStatus_EDMA_Success.

9.2.5.5 static edma_status_t EDMA_DRV_PrepDescriptorTransfer (*edma_chn_state_t * chn, edma_software_tcd_t * stcd, edma_transfer_config_t * config, bool enableInt, bool disableDmaRequest*) [inline], [static]

This function sets up the basic transfer for the descriptor. The minor loop setting is not used because the minor loop configuration impacts the global eDMA setting. The source minor loop offset is relevant to the destination minor loop offset. For these reasons, the minor loop offset configuration is treated as an advanced configuration. The user can call the [EDMA_HAL_STCDSetMinorLoopOffset\(\)](#) function to configure the minor loop offset feature.

Parameters

<i>channel</i>	Virtual channel number.
<i>chn</i>	The pointer to the channel state structure.
<i>stcd</i>	The pointer to the descriptor.
<i>config</i>	Configuration for the basic transfer.
<i>enableInt</i>	Enables (true) or Disables (false) interrupt on TCD complete.
<i>disableDmaRequest</i>	Disables (true) or Enable (false) DMA request on TCD complete.

9.2.5.6 static edma_status_t EDMA_DRV_PrepDescriptorScatterGather (*edma_software_tcd_t * stcd, edma_software_tcd_t * nextStcd*) [inline], [static]

This function enables the scatter/gather feature for the software TCD and configures the next TCD address. This address points to the beginning of a 0-modulo-32 byte region containing the next transfer TCD to be loaded into this channel. The channel reload is performed as the major iteration count completes. The scatter/gather address must be 0-modulo-32-byte. Otherwise, a configuration error is reported.

eDMA Peripheral Driver

Parameters

<i>stcd</i>	The pointer to the software TCD, which needs to link to the software TCD. The address needs to be aligned to 32 bytes.
<i>nextStcd</i>	The pointer to the software TCD, which is to be linked to the software TCD. The address needs to be aligned to 32 bytes.

9.2.5.7 static edma_status_t EDMA_DRV_PrepDescriptorChannelLink (edma_software_tcd_t * *stcd*, uint32_t *linkChn*) [inline], [static]

If the major link is enabled, after the major loop counter is exhausted, the eDMA engine initiates a channel service request at the channel defined by these six bits by setting that channel start bits.

Parameters

<i>stcd</i>	The pointer to the software TCD. The address need to be aligned to 32 bytes.
<i>linkChn</i>	Channel number for major link

9.2.5.8 edma_status_t EDMA_DRV_PushDescriptorToReg (edma_chn_state_t * *chn*, edma_software_tcd_t * *stcd*)

Parameters

<i>chn</i>	The pointer to the channel state structure.
<i>stcd</i>	memory pointing to the software TCD.

9.2.5.9 edma_status_t EDMA_DRV_ConfigLoopTransfer (edma_chn_state_t * *chn*, edma_software_tcd_t * *stcd*, edma_transfer_type_t *type*, uint32_t *srcAddr*, uint32_t *destAddr*, uint32_t *size*, uint32_t *bytesOnEachRequest*, uint32_t *totalLength*, uint8_t *number*)

This function configures the descriptors in a loop chain. The user passes a block of memory into this function and the memory is divided into the "period" sub blocks. The DMA driver configures the "period" descriptors. Each descriptor stands for a sub block. The DMA driver transfers data from the first descriptor to the last descriptor. Then, the DMA driver wraps to the first descriptor to continue the loop. The interrupt handler is called every time a descriptor is completed. The user can get a transfer status of a descriptor by calling the `edma_get_descriptor_status()` function in the interrupt handler or any other task context. At the same time, calling the `edma_update_descriptor()` function notifies the DMA driver that the content belonging to a descriptor is already updated and the DMA needs to count it as and underflow next time it loops to this descriptor.

Parameters

<i>chn</i>	The pointer to the channel state structure.
<i>stcd</i>	Memory pointing to software TCDs. The user must prepare this memory block. The required memory size is equal to a "period" * size of(edma_software_tcd_t). At the same time, the "stcd" must align with 32 bytes. If not, an error occurs in the eDMA driver.
<i>type</i>	Transfer type.
<i>srcAddr</i>	A source register address or a start memory address.
<i>destAddr</i>	A destination register address or a start memory address.
<i>size</i>	Size to be transferred on every DMA write/read. Source/Dest share the same write/read size.
<i>bytesOnEachRequest</i>	Size write/read for every trigger of the DMA request.
<i>totalLength</i>	Total length of Memory.
<i>number</i>	A number of the descriptor that is configured for this transfer configuration.

Returns

An error code of kStatus_EDMA_Success

9.2.5.10 `edma_status_t EDMA_DRV_ConfigScatterGatherTransfer (edma_chn_state_t * chn, edma_software_tcd_t * stcd, edma_transfer_type_t type, uint32_t size, uint32_t bytesOnEachRequest, edma_scatter_gather_list_t * srcList, edma_scatter_gather_list_t * destList, uint8_t number)`

This function configures the descriptors into a sing-end chain. The user passes blocks of memory into this function. The interrupt is triggered only when the last memory block is completed. The memory block information is passed with the `edma_scatter_gather_list_t` data structure, which can tell the memory address and length. The DMA driver configures the descriptor for each memory block, transfers the descriptor from the first one to the last one, and stops.

Parameters

<i>chn</i>	The pointer to the channel state structure.
------------	---

eDMA Peripheral Driver

<i>stcd</i>	Memory pointing to software TCDs. The user must prepare this memory block. The required memory size is equal to the "number" * size of(edma_software_tcd_t). At the same time, the "stcd" must align with 32 bytes. If not, an error occurs in the eDMA driver.
<i>type</i>	Transfer type.
<i>size</i>	Size to be transferred on each DMA write/read. Source/Dest share the same write/read size.
<i>bytesOnEachRequest</i>	Size write/read for each trigger of the DMA request.
<i>srcList</i>	Data structure storing the address and length to be transferred for source memory blocks. If the source memory is peripheral, the length is not used.
<i>destList</i>	Data structure storing the address and length to be transferred for dest memory blocks. If in the memory-to-memory transfer mode, the user must ensure that the length of the dest scatter gather list is equal to the source scatter gather list. If the dest memory is a peripheral register, the length is not used.
<i>number</i>	A number of memory block contained in the scatter gather list.

Returns

An error code of kStatus_EDMA_Success

9.2.5.11 **edma_status_t EDMA_DRV_StartChannel (edma_chn_state_t * *chn*)**

This function enables the eDMA channel DMA request.

Parameters

<i>chn</i>	The pointer to the channel state structure.
------------	---

Returns

An eDMA error codes or kStatus_EDMA_Success.

9.2.5.12 **edma_status_t EDMA_DRV_StopChannel (edma_chn_state_t * *chn*)**

This function disables the eDMA channel DMA request.

Parameters

<i>chn</i>	The pointer to the channel state structure.
------------	---

Returns

An eDMA error codes or kStatus_EDMA_Success.

9.2.5.13 **edma_status_t EDMA_DRV_InstallCallback (edma_chn_state_t * *chn*, edma_callback_t *callback*, void * *parameter*)**

This function register the callback function and the parameter into the eDMA channel state structure. The callback function is called when the channel is complete or a channel error occurs. The eDMA driver passes the channel status to this callback function to indicate whether it is caused by the channel complete event or the channel error event.

To un-register the callback function, the user can set the callback function to "NULL" and call this function.

Parameters

<i>chn</i>	The pointer to the channel state structure.
<i>callback</i>	The pointer to the callback function.
<i>parameter</i>	The pointer to the callback function's parameter.

Returns

An eDMA error codes or kStatus_EDMA_Success.

9.2.5.14 **void EDMA_DRV_IRQHandler (uint8_t *channel*)**

This function is provided as the default flow for eDMA channel interrupt. This function clears status, and calls the callback functions. The user can add this function into the hardware interrupt entry can implement a custom interrupt action function.

Parameters

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

9.2.5.15 **void EDMA_DRV_ErrorIRQHandler (uint8_t *instance*)**

This function is provided as the default action for eDMA module error interrupt. This function clears status, stops the error on a eDMA channel , and calls the eDMA channel callback function if the error

eDMA Peripheral Driver

eDMA channel is already requested. The user can add this function into the eDMA error interrupt entry and implement a custom interrupt action function.

Parameters

<i>instance</i>	eDMA module indicator.
-----------------	------------------------

9.2.5.16 static edma_chn_status_t EDMA_DRV_GetChannelStatus (edma_chn_state_t * *chn*) [inline], [static]

Parameters

<i>chn</i>	The pointer to the channel state structure.
------------	---

Returns

Channel status.

9.2.5.17 static uint32_t EDMA_DRV_GetFinishedBytes (edma_chn_state_t * *chn*) [inline], [static]

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the left bytes not to be transferred to the user. This function can't be used for a multi-TCD scenario and can only be used for one TCD scenario.

Parameters

<i>chn</i>	The pointer to the channel state structure.
------------	---

Returns

Bytes already transferred for current TCD.

9.3 eDMA request

This chapter describes the programming interface of the eDMA DMA request resource.

Chapter 10

Ethernet MAC (ENET)

The Kinetis SDK provides both HAL and Peripheral drivers for the Ethernet (ENET) block of Kinetis devices.

Modules

- ENET HAL driver

This part describes the programming interface of the ENET HAL driver.

- ENET Peripheral Driver

This part describes the programming interface of the ENET Peripheral Driver.

- ENET Physical Layer Driver

This part describes the programming interface of the ENET Physical Layer Driver.

- ENET RTCS Adaptor

This part describes the programming interface of the ENET RTCS Adaptor.

10.1 ENET HAL driver

This chapter describes the programming interface of the ENET HAL driver.

Data Structures

- struct `enet_bd_struct_t`
Defines the buffer descriptor structure for the little-Endian system and endianness configurable IP. [More...](#)
- struct `enet_config_rmii_t`
Defines the RMII/MII configuration structure. [More...](#)
- struct `enet_config_ptp_timer_t`
Defines the configuration structure for the 1588 PTP timer. [More...](#)
- struct `enet_config_tx_accelerator_t`
Defines the transmit accelerator configuration. [More...](#)
- struct `enet_config_rx_accelerator_t`
Defines the receive accelerator configuration. [More...](#)
- struct `enet_config_tx_fifo_t`
Defines the transmit FIFO configuration. [More...](#)
- struct `enet_config_rx_fifo_t`
Defines the receive FIFO configuration. [More...](#)
- struct `enet_mib_rx_stat_t`
@ brief Defines the receive statistics of MIB [More...](#)
- struct `enet_mib_tx_stat_t`
@ brief Defines the transmit statistics of MIB [More...](#)
- struct `enet_mac_config_t`
Defines the basic configuration structure for the ENET device. [More...](#)

Macros

- `#define SYSTEM_LITTLE_ENDIAN (1)`
Defines the system endian type.
- `#define BSWAP_16(x) ((uint16_t)((uint16_t)((uint16_t)(x) & (uint16_t)0xFF00) >> 0x8) | (uint16_t)((uint16_t)(x) & (uint16_t)0xFF) << 0x8))`
Define macro to do the endianness swap.
- `#define ENET_ALIGN(x, align) (((unsigned int)((x) + ((align)-1)) & (unsigned int)(~(unsigned int)((align)- 1)))`
Defines the alignment operation.

Enumerations

- enum `enet_status_t` { ,

kStatus_ENET_InvalidInput,

kStatus_ENET_InvalidDevice,

kStatus_ENET_InitTimeout,

kStatus_ENET_MemoryAllocateFail,

kStatus_ENET_GetClockFreqFail,

kStatus_ENET_Initialized,

kStatus_ENET_Open,

kStatus_ENET_Close,

kStatus_ENET_Layer2UnInitialized,

kStatus_ENET_Layer2OverLarge,

kStatus_ENET_Layer2BufferFull,

kStatus_ENET_Layer2TypeError,

kStatus_ENET_PtpringBufferFull,

kStatus_ENET_PtpringBufferEmpty,

kStatus_ENET_SMIUninitialized,

kStatus_ENET_SMIVisitTimeout,

kStatus_ENET_RxbdInvalid,

kStatus_ENET_RxbdEmpty,

kStatus_ENET_RxbdTrunc,

kStatus_ENET_RxbdError,

kStatus_ENET_RxBdFull,

kStatus_ENET_SmallRxBuffSize,

kStatus_ENET_NoEnoughRxBuffers,

kStatus_ENET_LargeBufferFull,

kStatus_ENET_TxLarge,

kStatus_ENET_TxbdFull,

kStatus_ENET_TxbdNull,

kStatus_ENET_TxBufferNull,

kStatus_ENET_NoRxBufferLeft,

kStatus_ENET_UnknownCommand,

kStatus_ENET_TimeOut,

kStatus_ENET_MulticastPointerNull,

kStatus_ENET_NoMulticastAddr,

kStatus_ENET_AlreadyAddedMulticast,

kStatus_ENET_PHYAutoDiscoverFail }
- Defines the Status return codes.*
- enum `enet_rx_bd_control_status_t` {

ENET HAL driver

```
kEnetRxBdEmpty = 0x8000U,  
kEnetRxBdRxSoftOwner1 = 0x4000U,  
kEnetRxBdWrap = 0x2000U,  
kEnetRxBdRxSoftOwner2 = 0x1000U,  
kEnetRxBdLast = 0x0800U,  
kEnetRxBdMiss = 0x0100U,  
kEnetRxBdBroadCast = 0x0080U,  
kEnetRxBdMultiCast = 0x0040U,  
kEnetRxBdLengthViolation = 0x0020U,  
kEnetRxBdNoOctet = 0x0010U,  
kEnetRxBdCrc = 0x0004U,  
kEnetRxBdOverRun = 0x0002U,  
kEnetRxBdTrunc = 0x0001U }
```

Defines the control and status region of the receive buffer descriptor.

- enum `enet_rx_bd_control_extend0_t` {
 kEnetRxBdIpv4 = 0x0001U,
 kEnetRxBdIpv6 = 0x0002U,
 kEnetRxBdVlan = 0x0004U,
 kEnetRxBdProtocolChecksumErr = 0x0010U,
 kEnetRxBdIpHeaderChecksumErr = 0x0020U }

Defines the control extended region1 of the receive buffer descriptor.

- enum `enet_rx_bd_control_extend1_t` {
 kEnetRxBdIntrrupt = 0x0080U,
 kEnetRxBdUnicast = 0x0100U,
 kEnetRxBdCollision = 0x0200U,
 kEnetRxBdPhyErr = 0x0400U,
 kEnetRxBdMacErr = 0x8000U }

Defines the control extended region2 of the receive buffer descriptor.

- enum `enet_tx_bd_control_status_t` {
 kEnetTxBdReady = 0x8000U,
 kEnetTxBdTxSoftOwner1 = 0x4000U,
 kEnetTxBdWrap = 0x2000U,
 kEnetTxBdTxSoftOwner2 = 0x1000U,
 kEnetTxBdLast = 0x0800U,
 kEnetTxBdTransmitCrc = 0x0400U }

Defines the control status of the transmit buffer descriptor.

- enum `enet_tx_bd_control_extend0_t` {
 kEnetTxBdTxErr = 0x8000U,
 kEnetTxBdTxUnderFlowErr = 0x2000U,
 kEnetTxBdExcessCollisionErr = 0x1000U,
 kEnetTxBdTxFrameErr = 0x0800U,
 kEnetTxBdLatecollisionErr = 0x0400U,
 kEnetTxBdOverFlowErr = 0x0200U,
 kEnetTxTimestampErr = 0x0100U }

Defines the control extended region1 of the transmit buffer descriptor.

- enum `enet_tx_bd_control_extend1_t` {

```
kEnetTxBdTxInterrupt = 0x4000U,
kEnetTxBdTimeStamp = 0x2000U }
```

Defines the control extended region2 of the transmit buffer descriptor.

- enum `enet_constant_parameter_t` {

kEnetMacAddrLen = 6U,
 kEnetHashValMask = 0x1FU,
 kEnetMinBuffSize = 256U,
 kEnetMaxTimeout = 0xFFFFU,
 kEnetMdcFreq = 2500000U }

Defines the macro to the different ENET constant value.

- enum `enet_fifo_configure_t` {

kEnetMinTxFifoAlmostFull = 6U,
 kEnetMinFifoAlmostEmpty = 4U,
 kEnetDefaultTxFifoAlmostFull = 8U }

Defines the normal fifo configuration for ENET MAC.

- enum `enet_mac_operate_mode_t` {

kEnetMacNormalMode = 0U,
 kEnetMacSleepMode = 1U }

Defines the normal operating mode and sleep mode for ENET MAC.

- enum `enet_config_rmii_mode_t` {

kEnetCfgMii = 0U,
 kEnetCfgRmii = 1U }

Defines the RMII or MII mode for data interface between the MAC and the PHY.

- enum `enet_config_speed_t` {

kEnetCfgSpeed100M = 0U,
 kEnetCfgSpeed10M = 1U }

Defines the 10 Mbps or 100 Mbps speed mode for the data transfer.

- enum `enet_config_duplex_t` {

kEnetCfgHalfDuplex = 0U,
 kEnetCfgFullDuplex = 1U }

Defines the half or full duplex mode for the data transfer.

- enum `enet_mii_write_t` {

kEnetWriteNoCompliant = 0U,
 kEnetWriteValidFrame = 1U }

Defines the write operation for the MII.

- enum `enet_mii_read_t` {

kEnetReadValidFrame = 2U,
 kEnetReadNoCompliant = 3U }

Defines the read operation for the MII.

- enum `enet_mdio_holdon_clkcycle_t` {

kEnetMdioHoldOneClkCycle = 0U,
 kEnetMdioHoldTwoClkCycle = 1U,
 kEnetMdioHoldThreeClkCycle = 2U,
 kEnetMdioHoldFourClkCycle = 3U,
 kEnetMdioHoldFiveClkCycle = 4U,
 kEnetMdioHoldSixClkCycle = 5U,
 kEnetMdioHoldSevenClkCycle = 6U,

ENET HAL driver

- ```
kEnetMdioHoldEightClkCycle = 7U }

 Define holdon time on MDIO output.
• enum enet_special_address_filter_t {
 kEnetSpecialAddressInit = 0U,
 kEnetSpecialAddressEnable = 1U,
 kEnetSpecialAddressDisable = 2U }

 Defines the initialization, enables or disables the operation for a special address filter.
• enum enet_timer_channel_t {
 kEnetTimerChannel1 = 0U,
 kEnetTimerChannel2 = 1U,
 kEnetTimerChannel3 = 2U,
 kEnetTimerChannel4 = 3U }

 Defines the 1588 timer channel numbers.
• enum enet_timer_channel_mode_t {
 kEnetChannelDisable = 0U,
 kEnetChannelRisingCapture = 1U,
 kEnetChannelFallingCapture = 2U,
 kEnetChannelBothCapture = 3U,
 kEnetChannelSoftCompare = 4U,
 kEnetChannelToggleCompare = 5U,
 kEnetChannelClearCompare = 6U,
 kEnetChannelSetCompare = 7U,
 kEnetChannelClearCompareSetOverflow = 10U,
 kEnetChannelSetCompareClearOverflow = 11U,
 kEnetChannelPulseLowonCompare = 14U,
 kEnetChannelPulseHighonCompare = 15U }

 Defines the capture or compare mode for 1588 timer channels.
• enum enet_interrupt_request_t {
 kEnetBabrInterrupt = 0x40000000U,
 kEnetBabtInterrupt = 0x20000000U,
 kEnetGraceStopInterrupt = 0x10000000U,
 kEnetTxFrameInterrupt = 0x08000000U,
 kEnetTxByteInterrupt = 0x04000000U,
 kEnetRxFrameInterrupt = 0x02000000U,
 kEnetRxByteInterrupt = 0x01000000U,
 kEnetMiiInterrupt = 0x00800000U,
 kEnetEBusERInterrupt = 0x00400000U,
 kEnetLateCollisionInterrupt = 0x00200000U,
 kEnetRetryLimitInterrupt = 0x00100000U,
 kEnetUnderrunInterrupt = 0x00080000U,
 kEnetPayloadRxInterrupt = 0x00040000U,
 kEnetWakeupInterrupt = 0x00020000U,
 kEnetTsAvailInterrupt = 0x00010000U,
 kEnetTsTimerInterrupt = 0x00008000U,
 kEnetAllInterrupt = 0x7FFFFFFFU }

 Defines the RXFRAME/RXBYTE/TXFRAME/TXBYTE/MII/TSTIMER/TSAVAIL interrupt source for ENE-
```

- enum `enet_irq_number_t` {
   
    `kEnetTsTimerNumber` = 0,
   
    `kEnetReceiveNumber` = 1,
   
    `kEnetTransmitNumber` = 2,
   
    `kEnetMiiErrorNumber` = 3 }
- enum `enet_frame_max_t` {
   
    `kEnetMaxFrameSize` = 1518,
   
    `kEnetMaxFrameVlanSize` = 1522,
   
    `kEnetMaxFrameDateSize` = 1500 ,
   
    `kEnetDefaultIpg` = 12,
   
    `kEnetMaxFrameBdNumbers` = 7,
   
    `kEnetFrameFcsLen` = 4,
   
    `kEnetEthernetHeadLen` = 14,
   
    `kEnetEthernetVlanHeadLen` = 18 }
   
*Defines the ENET main constant.*
- enum `enet_mac_control_flag_t` {
   
    `kEnetStopModeEnable` = 0x1U ,
   
    `kEnetPayloadlenCheckEnable` = 0x4U,
   
    `kEnetRxFlowControlEnable` = 0x8U,
   
    `kEnetRxCrcFwdEnable` = 0x10U,
   
    `kEnetRxPauseFwdEnable` = 0x20U,
   
    `kEnetRxPadRemoveEnable` = 0x40U,
   
    `kEnetRxBcRejectEnable` = 0x80U,
   
    `kEnetRxPromiscuousEnable` = 0x100U,
   
    `kEnetTxCrcFwdEnable` = 0x200U,
   
    `kEnetTxCrcBdEnable` = 0x400U,
   
    `kEnetMacAddrInsert` = 0x800U,
   
    `kEnetTxAccelEnable` = 0x1000U,
   
    `kEnetRxAccelEnable` = 0x2000U,
   
    `kEnetStoreAndFwdEnable` = 0x4000U,
   
    `kEnetMacMibEnable` = 0x8000U,
   
    `kEnetSMIPreambleDisable` = 0x10000U,
   
    `kEnetVlanTagEnabled` = 0x20000U,
   
    `kEnetMacEnhancedEnable` = 0x40000U }
   
*Defines the ENET MAC control Configure.*

## Functions

- `uint32_t ENET_HAL_Init (uint32_t baseAddr)`
  
*Initializes the ENET module to reset status.*
- `void ENET_HAL_SetMac (uint32_t baseAddr, const enet_mac_config_t *macCfgPtr, uint32_t sysClk)`
  
*Configures the Mac controller of the ENET device.*
- `void ENET_HAL_SetTxBuffDescriptors (uint32_t baseAddr, volatile enet_bd_struct_t *txBds,`

## ENET HAL driver

- ```
uint8_t *txBuffer, uint32_t txBdNumber, uint32_t txBuffSizeAlign)
    Configures the ENET transmit buffer descriptors.
• void ENET\_HAL\_SetRxBuffDescriptors (uint32_t baseAddr, volatile enet\_bd\_struct\_t *rxBds,
  uint8_t *rxBuffer, uint32_t rxBdNumber, uint32_t rxBuffSizeAlign)
    Configures the ENET receive buffer descriptors.
• void ENET\_HAL\_SetFifo (uint32_t baseAddr, const enet\_mac\_config\_t *macCfgPtr)
    Configures the transmit and receive FIFO of the ENET device.
• void ENET\_HAL\_GetMibRxStat (uint32_t baseAddr, enet\_mib\_rx\_stat\_t *rxStat)
    Gets all received statistics from MIB.
• void ENET\_HAL\_GetMibTxStat (uint32_t baseAddr, enet\_mib\_tx\_stat\_t *txStat)
    Gets all transmitted statistics from MIB.
• static void ENET\_HAL\_SetStopCmd (uint32_t baseAddr, bool enable)
    Enables or disables the stop enable signal control.
• static void ENET\_HAL\_SetDebugCmd (uint32_t baseAddr, bool enable)
    Enables or disables Mac entering the hardware freeze mode when the device enters debug mode.
• void ENET\_HAL\_SetMacMode (uint32_t baseAddr, enet\_mac\_operate\_mode\_t mode)
    Configures the Mac operating mode.
• void ENET\_HAL\_SetMacAddr (uint32_t baseAddr, uint8_t *hwAddr)
    Sets the Mac address.
• static void ENET\_HAL\_SetMacAddrInsertCmd (uint32_t baseAddr, bool enable)
    Enables or disables Mac address modification on transmit.
• void ENET\_HAL\_SetMulticastAddrHash (uint32_t baseAddr, uint32_t crcValue, enet\_special\_address\_filter\_t mode)
    Sets the hardware addressing filtering to a multicast group address.
• void ENET\_HAL\_SetUnicastAddrHash (uint32_t baseAddr, uint32_t crcValue, enet\_special\_address\_filter\_t mode)
    Sets the hardware addressing filtering to an individual address.
• static void ENET\_HAL\_SetTxcrcFwdCmd (uint32_t baseAddr, bool enable)
    Enables/disables the forwarding frame from an application with the CRC for the transmitted frames.
• static void ENET\_HAL\_SetRxcrcFwdCmd (uint32_t baseAddr, bool enable)
    Enables/disables forward the CRC field of the received frame.
• static void ENET\_HAL\_SetPauseFwdCmd (uint32_t baseAddr, bool enable)
    Enables/disables pause frames forwarding.
• static void ENET\_HAL\_SetPadRemoveCmd (uint32_t baseAddr, bool enable)
    Enables/disables frame padding remove on receive.
• static void ENET\_HAL\_SetFlowControlCmd (uint32_t baseAddr, bool enable)
    Enables/disables the flow control.
• static void ENET\_HAL\_SetBroadcastRejectCmd (uint32_t baseAddr, bool enable)
    Enables/disables the broadcast frame reject.
• static void ENET\_HAL\_SetPayloadCheckCmd (uint32_t baseAddr, bool enable)
    Enables/disables the payload length check.
• static void ENET\_HAL\_SetGraceTxStopCmd (uint32_t baseAddr, bool enable)
    Enables/disables the graceful transmit stop.
• static void ENET\_HAL\_SetPauseDuration (uint32_t baseAddr, uint32_t pauseDuration)
    Sets the pause duration for the pause frame.
• static void ENET\_HAL\_SetTxPauseCmd (uint32_t baseAddr, bool enable)
    Configures the pause duration field and transmits the pause frame.
• static bool ENET\_HAL\_GetTxPause (uint32_t baseAddr)
    Gets the transmit pause frame status.
• static bool ENET\_HAL\_GetRxPause (uint32_t baseAddr)
    Gets the receive pause frame status.
```

- static void [ENET_HAL_SetTxInterPacketGap](#) (uint32_t baseAddr, uint32_t ipgValue)

Sets the transmit inter-packet gap.
- static void [ENET_HAL_SetTruncLen](#) (uint32_t baseAddr, uint32_t length)

Sets the receive frame truncation length.
- static void [ENET_HAL_SetRxMaxFrameLen](#) (uint32_t baseAddr, uint32_t maxFrameSize)

Sets the maximum receive frame length.
- static void [ENET_HAL_SetRxMaxBuffSize](#) (uint32_t baseAddr, uint32_t maxBufferSize)

Sets the maximum receive buffer size for receive buffer descriptor.
- void [ENET_HAL_SetTxFifo](#) (uint32_t baseAddr, [enet_config_tx_fifo_t](#) *thresholdCfg)

Configures the ENET transmit FIFO.
- void [ENET_HAL_SetRxFifo](#) (uint32_t baseAddr, [enet_config_rx_fifo_t](#) *thresholdCfg)

Configures the ENET receive FIFO.
- static void [ENET_HAL_SetRxBuffDescripAddr](#) (uint32_t baseAddr, uint32_t rxBdAddr)

Sets the start address for ENET receive buffer descriptors.
- static void [ENET_HAL_SetTxBuffDescripAddr](#) (uint32_t baseAddr, uint32_t txBdAddr)

Sets the start address for ENET transmit buffer descriptors.
- void [ENET_HAL_InitRxBuffDescriptors](#) (volatile [enet_bd_struct_t](#) *rxBds, uint8_t *rxBuff, uint32_t rxbdNum, uint32_t rxBuffSizeAlign)

Initializes receive buffer descriptors.
- void [ENET_HAL_InitTxBuffDescriptors](#) (volatile [enet_bd_struct_t](#) *txBds, uint8_t *txBuff, uint32_t txbdNum, uint32_t txBuffSizeAlign)

Initializes transmit buffer descriptors.
- void [ENET_HAL_UpdateRxBuffDescriptor](#) (volatile [enet_bd_struct_t](#) *rxBds, uint8_t *data, bool isbufferUpdate)

Updates the receive buffer descriptor.
- void [ENET_HAL_UpdateTxBuffDescriptor](#) (volatile [enet_bd_struct_t](#) *txBds, uint16_t length, bool isTxtsCfged, bool isTxCrcEnable, bool isLastOne)

Updates the transmit buffer descriptor.
- static void [ENET_HAL_ClearTxBuffDescriptor](#) (volatile [enet_bd_struct_t](#) *curBd)

Clears the context in the transmit buffer descriptors.
- static uint16_t [ENET_HAL_GetRxBuffDescripControl](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the control and the status region of the receive buffer descriptors.
- static uint16_t [ENET_HAL_GetTxBuffDescripControl](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the control and the status region of the transmit buffer descriptors.
- static uint16_t [ENET_HAL_GetRxbuffDescripExtControlOne](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the extended control region one of the receive buffer descriptor.
- static uint16_t [ENET_HAL_GetRxbuffDescripExtControlTwo](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the extended control region two of the receive buffer descriptor.
- static uint16_t [ENET_HAL_GetTxBuffDescripExtControl](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the extended control region of the transmit buffer descriptors.
- static bool [ENET_HAL_GetTxBuffDescripTsFlag](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the transmit buffer descriptor timestamp flag.
- uint16_t [ENET_HAL_GetBuffDescripLen](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the data length of the buffer descriptors.
- uint8_t * [ENET_HAL_GetBuffDescripData](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the buffer address of the buffer descriptors.
- uint32_t [ENET_HAL_GetBuffDescripTs](#) (volatile [enet_bd_struct_t](#) *curBd)

Gets the timestamp of the buffer descriptors.
- static void [ENET_HAL_SetRxBuffDescripActive](#) (uint32_t baseAddr)

Activates the receive buffer descriptor.
- static void [ENET_HAL_SetTxBuffDescripActive](#) (uint32_t baseAddr)

ENET HAL driver

- void **ENET_HAL_SetRMIIMode** (uint32_t baseAddr, [enet_config_rmii_t](#) *rmiiCfgPtr)
Configures the (R)MII data interface of ENET.
- static void **ENET_HAL_SetRMIIISpeed** (uint32_t baseAddr, [enet_config_speed_t](#) speed)
Configures the (R)MII speed of ENET.
- static void **ENET_HAL_SetSMI** (uint32_t baseAddr, uint32_t miiSpeed, [enet_mdio_holdon_clkcycle_t](#) clkCycle, bool isPreambleDisabled)
Configures the SMI(serial Management interface) of ENET.
- static bool **ENET_HAL_GetSMI** (uint32_t baseAddr)
Gets SMI configuration status.
- static uint32_t **ENET_HAL_GetSMIData** (uint32_t baseAddr)
Reads data from PHY.
- void **ENET_HAL_SetSMIRead** (uint32_t baseAddr, uint32_t phyAddr, uint32_t phyReg, [enet_mii_read_t](#) operation)
Sets the SMI(serial Management interface) read command.
- void **ENET_HAL_SetSMIWrite** (uint32_t baseAddr, uint32_t phyAddr, uint32_t phyReg, [enet_mii_write_t](#) operation, uint32_t data)
Sets the SMI(serial Management interface) write command.
- static void **ENET_HAL_Enable** (uint32_t baseAddr)
Enables the ENET module.
- static void **ENET_HAL_Disable** (uint32_t baseAddr)
Disables the ENET module.
- static void **ENET_HAL_SetEnhancedMacCmd** (uint32_t baseAddr, bool enable)
Enables or disables the enhanced functionality of the MAC(1588 feature).
- void **ENET_HAL_SetIntMode** (uint32_t baseAddr, [enet_interrupt_request_t](#) source, bool enable)
Enables/Disables the ENET interrupt.
- static void **ENET_HAL_ClearIntStatusFlag** (uint32_t baseAddr, [enet_interrupt_request_t](#) source)
Clears ENET interrupt events.
- static bool **ENET_HAL_GetIntStatusFlag** (uint32_t baseAddr, [enet_interrupt_request_t](#) source)
Gets the ENET interrupt status.
- static void **ENET_HAL_SetPromiscuousCmd** (uint32_t baseAddr, bool enable)
Enables/disables the ENET promiscuous mode.
- static void **ENET_HAL_SetMibClearCmd** (uint32_t baseAddr, bool enable)
Enables/disables the clear MIB counter.
- static void **ENET_HAL_SetMibCmd** (uint32_t baseAddr, bool enable)
Sets the enable/disable of the MIB block.
- static bool **ENET_HAL_GetMibStatus** (uint32_t baseAddr)
Gets the MIB idle status.
- void **ENET_HAL_SetTxAccelerator** (uint32_t baseAddr, [enet_config_tx_accelerator_t](#) *txCfgPtr)
Sets the transmit accelerator.
- void **ENET_HAL_SetRxAccelerator** (uint32_t baseAddr, [enet_config_rx_accelerator_t](#) *rxCfgPtr)
Sets the receive accelerator.
- void **ENET_HAL_Set1588TimerStart** (uint32_t baseAddr, [enet_config_ptp_timer_t](#) *ptpCfgPtr)
Configures the 1588 timer and run the 1588 timer.
- void **ENET_HAL_Set1588TimerChnCmp** (uint32_t baseAddr, [enet_timer_channel_t](#) channel, uint32_t cmpValOld, uint32_t cmpValNew)
Configures the 1588 timer channel compare feature.
- void **ENET_HAL_Set1588Timer** (uint32_t baseAddr, [enet_config_ptp_timer_t](#) *ptpCfgPtr)
Initializes the 1588 timer.
- static void **ENET_HAL_Set1588TimerCmd** (uint32_t baseAddr, uint32_t enable)
Enables or disables the 1588 PTP timer.

- static void [ENET_HAL_Set1588TimerRestart](#) (uint32_t baseAddr)
Restarts the 1588 timer.
- static void [ENET_HAL_Set1588TimerAdjust](#) (uint32_t baseAddr, uint32_t increaseCorrection, uint32_t periodCorrection)
Adjusts the 1588 timer.
- static void [ENET_HAL_Set1588TimerCapture](#) (uint32_t baseAddr)
Sets the capture command to the 1588 timer.
- static void [ENET_HAL_Set1588TimerNewTime](#) (uint32_t baseAddr, uint32_t nanSecond)
Sets the 1588 timer.
- static uint32_t [ENET_HAL_Get1588TimerCurrentTime](#) (uint32_t baseAddr)
Gets the time from the 1588 timer.
- static void [ENET_HAL_Set1588TimerChnMode](#) (uint32_t baseAddr, enet_timer_channel_t channel, enet_timer_channel_mode_t mode)
Initializes one channel for the four-channel 1588 timer.
- static void [ENET_HAL_Set1588TimerChnInt](#) (uint32_t baseAddr, enet_timer_channel_t channel, bool enable)
Sets the 1588 time channel interrupt.
- static void [ENET_HAL_Set1588TimerChnCmpVal](#) (uint32_t baseAddr, enet_timer_channel_t channel, uint32_t compareValue)
Sets the compare value for the 1588 timer channel.
- static bool [ENET_HAL_Get1588TimerChnStatus](#) (uint32_t baseAddr, enet_timer_channel_t channel)
Gets the 1588 timer channel status.
- static void [ENET_HAL_Clear1588TimerChnFlag](#) (uint32_t baseAddr, enet_timer_channel_t channel)
Clears the 1588 timer channel interrupt flag.
- static uint32_t [ENET_HAL_GetTxTs](#) (uint32_t baseAddr)
Gets the transmit timestamp.
- static uint16_t [ENET_HAL_GetTxPackets](#) (uint32_t baseAddr)
Gets the transmit packet count statistic.
- static uint16_t [ENET_HAL_GetTxBroadCastPacket](#) (uint32_t baseAddr)
Gets the transmit broadcast packet statistic.
- static uint16_t [ENET_HAL_GetTxMultiCastPacket](#) (uint32_t baseAddr)
Gets the transmit multicast packet statistic.
- static uint16_t [ENET_HAL_GetTxCrcAlignErrorPacket](#) (uint32_t baseAddr)
Gets the transmit packets with CRC/Align error.
- static uint16_t [ENET_HAL_GetTxUnderSizePacket](#) (uint32_t baseAddr)
Gets the transmit packets less than 64 bytes and good CRC.
- static uint16_t [ENET_HAL_GetTxFragPacket](#) (uint32_t baseAddr)
Gets the transmit packets less than 64 bytes and bad CRC.
- static uint16_t [ENET_HAL_GetTxOverSizePacket](#) (uint32_t baseAddr)
Gets the transmit packets over than MAX_FL bytes and good CRC.
- static uint16_t [ENET_HAL_GetTxJabPacket](#) (uint32_t baseAddr)
Gets the transmit packets over than MAX_FL bytes and bad CRC.
- static uint16_t [ENET_HAL_GetTxCollisionPacket](#) (uint32_t baseAddr)
Gets the transmit collision packets.
- static uint16_t [ENET_HAL_GetTxByte64Packet](#) (uint32_t baseAddr)
Gets the transmit 64-byte packet statistic.
- static uint16_t [ENET_HAL_GetTxByte65to127Packet](#) (uint32_t baseAddr)
Gets the transmit 65-byte to 127-byte packet statistic.

ENET HAL driver

- static uint16_t [ENET_HAL_GetTxByte128to255Packet](#) (uint32_t baseAddr)
Gets the transmit packets 128-byte to 255-byte.
- static uint16_t [ENET_HAL_GetTxByte256to511Packet](#) (uint32_t baseAddr)
Gets the transmit packets 256-byte to 511-byte.
- static uint16_t [ENET_HAL_GetTxByte512to1023Packet](#) (uint32_t baseAddr)
Gets the transmit packets 512-byte to 1023-byte.
- static uint16_t [ENET_HAL_GetTxByte1024to2047Packet](#) (uint32_t baseAddr)
Gets the transmit packets 1024-byte to 2047-byte.
- static uint16_t [ENET_HAL_GetTxOverByte2048Packet](#) (uint32_t baseAddr)
Gets the transmit packets greater than 2048-byte.
- static uint32_t [ENET_HAL_GetTxOctets](#) (uint32_t baseAddr)
Gets the transmit octets.
- static uint16_t [ENET_HAL_GetTxFramesOk](#) (uint32_t baseAddr)
Gets the Frames transmitted OK.
- static uint16_t [ENET_HAL_GetTxFramesOneCollision](#) (uint32_t baseAddr)
Gets the Frames transmitted with single collision.
- static uint16_t [ENET_HAL_GetTxFramesMultiCollision](#) (uint32_t baseAddr)
Gets the frames transmitted with multiple collision.
- static uint16_t [ENET_HAL_GetTxFrameCarrSenseError](#) (uint32_t baseAddr)
Gets the frames transmitted with carrier sense error.
- static uint16_t [ENET_HAL_GetTxFramesDelay](#) (uint32_t baseAddr)
Gets the frames transmitted after deferral delay.
- static uint16_t [ENET_HAL_GetTxFramesLateCollision](#) (uint32_t baseAddr)
Gets the frames transmitted with late collision.
- static uint16_t [ENET_HAL_GetTxFramesExcessiveCollision](#) (uint32_t baseAddr)
Gets the frames transmitted with excessive collisions.
- static uint16_t [ENET_HAL_GetTxFramesMacError](#) (uint32_t baseAddr)
Gets the frames transmitted with the Tx FIFO underrun.
- static uint16_t [ENET_HAL_GetTxFramesPause](#) (uint32_t baseAddr)
Gets the transmitted flow control Pause Frames.
- static uint32_t [ENET_HAL_GetTxOctetFramesOk](#) (uint32_t baseAddr)
Gets the octet count for frames transmitted without error.
- static uint16_t [ENET_HAL_GetRxPackets](#) (uint32_t baseAddr)
Gets the receive packet count.
- static uint16_t [ENET_HAL_GetRxBroadCastPacket](#) (uint32_t baseAddr)
Gets the receive broadcast packet count.
- static uint16_t [ENET_HAL_GetRxMultiCastPacket](#) (uint32_t baseAddr)
Gets the receive multicast packet count.
- static uint16_t [ENET_HAL_GetRxCrcAlignErrorPacket](#) (uint32_t baseAddr)
Gets the receive packets with CRC/Align error.
- static uint16_t [ENET_HAL_GetRxUnderSizePacket](#) (uint32_t baseAddr)
Gets the receive packets less than 64-byte and good CRC.
- static uint16_t [ENET_HAL_GetRxOverSizePacket](#) (uint32_t baseAddr)
Gets the receive packets greater than MAX_FL and good CRC.
- static uint16_t [ENET_HAL_GetRxFragPacket](#) (uint32_t baseAddr)
Gets the receive packets less than 64-byte and bad CRC.
- static uint16_t [ENET_HAL_GetRxJabPacket](#) (uint32_t baseAddr)
Gets the receive packets greater than MAX_FL and bad CRC.
- static uint16_t [ENET_HAL_GetRxByte64Packet](#) (uint32_t baseAddr)
Gets the receive packets with 64-byte.
- static uint16_t [ENET_HAL_GetRxByte65to127Packet](#) (uint32_t baseAddr)
Gets the receive packets with 65-to-127 byte.

- static uint16_t **ENET_HAL_GetRxByte128to255Packet** (uint32_t baseAddr)

Gets the receive packets with 65-byte to 127-byte.
- static uint16_t **ENET_HAL_GetRxByte256to511Packet** (uint32_t baseAddr)

Gets the receive packets with 128-byte to 255-byte.
- static uint16_t **ENET_HAL_GetRxByte512to1023Packet** (uint32_t baseAddr)

Gets the receive packets with 256-byte to 511-byte.
- static uint16_t **ENET_HAL_GetRxByte1024to2047Packet** (uint32_t baseAddr)

Gets the receive packets with 512-byte to 1023-byte.
- static uint16_t **ENET_HAL_GetRxByte1024to2047Packet** (uint32_t baseAddr)

Gets the receive packets with 1024-byte to 2047-byte.
- static uint16_t **ENET_HAL_GetRxOverByte2048Packet** (uint32_t baseAddr)

Gets the receive packets greater than 2048-byte.
- static uint32_t **ENET_HAL_GetRxOctets** (uint32_t baseAddr)

Gets the receive octets.
- static uint16_t **ENET_HAL_GetRxFramesDrop** (uint32_t baseAddr)

Gets the receive Frames not counted correctly.
- static uint16_t **ENET_HAL_GetRxFramesOk** (uint32_t baseAddr)

Gets the Frames received OK.
- static uint16_t **ENET_HAL_GetRxFramesCrcError** (uint32_t baseAddr)

Gets the Frames received with CRC error.
- static uint16_t **ENET_HAL_GetRxFramesAlignError** (uint32_t baseAddr)

Gets the Frames received with Alignment error.
- static uint16_t **ENET_HAL_GetRxFramesMacError** (uint32_t baseAddr)

Gets the FIFO overflow count.
- static uint16_t **ENET_HAL_GetRxFramesFlowControl** (uint32_t baseAddr)

Gets the received flow control Pause frames.
- static uint32_t **ENET_HAL_GetRxOctetsFramesOk** (uint32_t baseAddr)

Gets the octet count for Frames received without Error.

10.1.1 Data Structure Documentation

10.1.1.1 struct enet_bd_struct_t

Data Fields

- uint16_t **length**

Buffer descriptor data length.
- uint16_t **control**

Buffer descriptor control.
- uint8_t * **buffer**

Data buffer pointer.
- uint16_t **controlExtend0**

Extend buffer descriptor control0.
- uint16_t **controlExtend1**

Extend buffer descriptor control1.
- uint16_t **payloadCheckSum**

Internal payload checksum.
- uint8_t **headerLength**

Header length.
- uint8_t **protocolType**

ENET HAL driver

- *Protocol type.*
 • `uint16_t controlExtend2`
 Extend buffer descriptor control2.
 • `uint32_t timestamp`
 Timestamp.

10.1.1.2 struct enet_config_rmii_t

Data Fields

- `enet_config_rmii_mode_t mode`
 RMII/MII mode.
- `enet_config_speed_t speed`
 100M/10M Speed
- `enet_config_duplex_t duplex`
 Full/Duplex mode.
- `bool isRxOnTxDisabled`
 Disable rx and tx.
- `bool isLoopEnabled`
 MII loop mode.

10.1.1.3 struct enet_config_ptp_timer_t

Data Fields

- `bool isSlaveEnabled`
 Master or slave PTP timer.
- `uint32_t clockIncease`
 Timer increase value each clock period.
- `uint32_t period`
 Timer period for generate interrupt event.

10.1.1.4 struct enet_config_tx_accelerator_t

Data Fields

- `bool isIpCheckEnabled`
 Insert IP header checksum.
- `bool isProtocolCheckEnabled`
 Insert protocol checksum.
- `bool isShift16Enabled`
 Tx FIFO shift-16.

10.1.1.5 struct enet_config_rx_accelerator_t

Data Fields

- bool `isIpcheckEnabled`
Discard with wrong IP header checksum.
- bool `isProtocolCheckEnabled`
Discard with wrong protocol checksum.
- bool `isMacCheckEnabled`
Discard with Mac layer errors.
- bool `isPadRemoveEnabled`
Padding removal for short IP frames.
- bool `isShift16Enabled`
Rx FIFO shift-16.

10.1.1.6 struct enet_config_tx_fifo_t

Data Fields

- bool `isStoreForwardEnabled`
Transmit FIFO store and forward.
- uint8_t `txFifoWrite`
Transmit FIFO write.
- uint8_t `txEmpty`
Transmit FIFO chapter empty threshold, default zero.
- uint8_t `txAlmostEmpty`
Transmit FIFO chapter almost empty threshold, The minimum value of 4 should be set.
- uint8_t `txAlmostFull`
Transmit FIFO chapter almost full threshold, The minmum value of 6 is required a recommended value of at least 8 should be set.

10.1.1.6.0.25 Field Documentation

10.1.1.6.0.25.1 uint8_t enet_config_tx_fifo_t::txFifoWrite

This should be set when `isStoreForwardEnabled` is false. this field indicates the number of bytes in step of 64 bytes written to the Tx FiFO before transmission of a frame begins

10.1.1.7 struct enet_config_rx_fifo_t

Data Fields

- uint8_t `rxFull`
Receive FIFO chapter full threshold, default zero.
- uint8_t `rxAlmostFull`
Receive FIFO chapter almost full threshold, The minimum value of 4 should be set.
- uint8_t `rxEmpty`
Receive FIFO chapter empty threshold, default zero.
- uint8_t `rxAlmostEmpty`

Receive FIFO chapter almost empty threshold, The minimum value of 4 should be set.

10.1.1.8 struct enet_mib_rx_stat_t

Data Fields

- uint16_t rxPackets
Receive packets.
- uint16_t rxBroadcastPackets
Receive broadcast packets.
- uint16_t rxMulticastPackets
Receive multicast packets.
- uint16_t rxCrcAlignErrorPackets
Receive packets with crc/align error.
- uint16_t rxUnderSizeGoodPackets
Receive packets undersize and good crc.
- uint16_t rxUnderSizeBadPackets
Receive packets undersize and bad crc.
- uint16_t rxOverSizeGoodPackets
Receive packets oversize and good crc.
- uint16_t rxOverSizeBadPackets
Receive packets oversize and bad crc.
- uint16_t rxByte64Packets
Receive packets 64-byte.
- uint16_t rxByte65to127Packets
Receive packets 65-byte to 127-byte.
- uint16_t rxByte128to255Packets
Receive packets 128-byte to 255-byte.
- uint16_t rxByte256to511Packets
Receive packets 256-byte to 511-byte.
- uint16_t rxByte512to1023Packets
Receive packets 512-byte to 1023-byte.
- uint16_t rxByte1024to2047Packets
Receive packets 1024-byte to 2047-byte.
- uint16_t rxByteOver2048Packets
Receive packets over 2048-byte.
- uint32_t rxOctets
Receive octets.
- uint32_t ieeeOctetsrxFrameOk
Receive octets of received Frames ok.
- uint16_t ieeerxFrameDrop
Receive Frames dropped.
- uint16_t ieeerxFrameOk
Receive Frames ok.
- uint16_t ieeerxFrameCrcErr
Receive Frames with crc error.
- uint16_t ieeetxFrameAlignErr
Receive Frames with align error.
- uint16_t ieeetxFrameMacErr
Receive Frames with mac error.

- `uint16_t ieeetxFramePause`
Receive flow control pause frames.

10.1.1.9 `struct enet_mib_tx_stat_t`

Data Fields

- `uint16_t txPackets`
Transmit packets.
- `uint16_t txBroadcastPackets`
Transmit broadcast packets.
- `uint16_t txMulticastPackets`
Transmit multicast packets.
- `uint16_t txCrcAlignErrorPackets`
Transmit packets with crc/align error.
- `uint16_t txUnderSizeGoodPackets`
Transmit packets undersize and good crc.
- `uint16_t txUnderSizeBadPackets`
Transmit packets undersize and bad crc.
- `uint16_t txOverSizeGoodPackets`
Transmit packets oversize and good crc.
- `uint16_t txOverSizeBadPackets`
Transmit packets oversize and bad crc.
- `uint16_t txCollision`
Transmit packets with collision.
- `uint16_t txByte64Packets`
Transmit packets 64-byte.
- `uint16_t txByte65to127Packets`
Transmit packets 65-byte to 127-byte.
- `uint16_t txByte128to255Packets`
Transmit packets 128-byte to 255-byte.
- `uint16_t txByte256to511Packets`
Transmit packets 256-byte to 511-byte.
- `uint16_t txByte512to1023Packets`
Transmit packets 512-byte to 1023-byte.
- `uint16_t txByte1024to2047Packets`
Transmit packets 1024-byte to 2047-byte.
- `uint16_t txByteOver2048Packets`
Transmit packets over 2048-byte.
- `uint32_t txOctets`
Transmit octets.
- `uint32_t ieeeOctetstxFrameOk`
Transmit octets of transmitted frames ok.
- `uint16_t ieeetxFrameOk`
Transmit frames ok.
- `uint16_t ieeetxFrameOneCollision`
Transmit frames with single collision.
- `uint16_t ieeetxFrameMultiCollision`
Transmit frames with multicast collision.
- `uint16_t ieeetxFrameLateCollision`

ENET HAL driver

- *Transmit frames with late collision.*
• `uint16_t ieeetxFrmاءExcCollision`
 Transmit frames with excessive collision.
- `uint16_t ieeetxFrameDelay`
 Transmit frames after deferral delay.
- `uint16_t ieeetxFrameMacErr`
 Transmit frames with MAC error.
- `uint16_t ieeetxFrameCarrSenseErr`
 Transmit frames with carrier sense error.
- `uint16_t ieeetxFramePause`
 Transmit flow control Pause frame.

10.1.1.10 struct enet_mac_config_t

Data Fields

- `enet_mac_operate_mode_t macMode`
 Mac Normal or sleep mode.
- `uint16_t rxMaxFrameLen`
 Receive maximum frame length.
- `uint16_t rxTruncLen`
 Receive truncate length.
- `uint16_t txInterPacketGap`
 Transmit inter-packet-gap.
- `uint16_t pauseDuration`
 Pause duration.
- `uint8_t * macAddr`
 MAC hardware address.
- `enet_config_rmii_t * rmiiCfgPtr`
 RMII configure mode.
- `enet_mdio_holdon_clkcycle_t clkCycle`
 SMI:MDIO hold on clock cycle.
- `enet_config_rx_accelerator_t * rxAccelerPtr`
 Receive accelerator configure.
- `enet_config_tx_accelerator_t * txAccelerPtr`
 Transmit accelerator configure.
- `enet_config_rx_fifo_t rxFifo`
 Receive fifo configuration.
- `enet_config_tx_fifo_t txFifo`
 Transmit fifo configuration.

10.1.1.10.0.26 Field Documentation

10.1.1.10.0.26.1 enet_mdio_holdon_clkcycle_t enet_mac_config_t::clkCycle

Mac control configure, it is recommended to use `enet_mac_control_flag_t` it is special control set for loop mode, sleep mode, crc forward/terminate etc

10.1.2 Macro Definition Documentation

10.1.2.1 #define SYSTEM_LITTLE_ENDIAN (1)

10.1.2.2 #define ENET_ALIGN(x, align) ((unsigned int)((x) + ((align)-1)) & (unsigned int)(~(unsigned int)((align)- 1)))

10.1.3 Enumeration Type Documentation

10.1.3.1 enum enet_status_t

Enumerator

kStatus_ENET_InvalidInput Invalid ENET input parameter.
kStatus_ENET_InvalidDevice Invalid ENET device.
kStatus_ENET_InitTimeout ENET initialize timeout.
kStatus_ENET_MemoryAllocateFail Memory allocate failure.
kStatus_ENET_GetClockFreqFail Get clock frequency failure.
kStatus_ENET_Initialized ENET device already initialized.
kStatus_ENET_Open Open ENET device.
kStatus_ENET_Close Close ENET device.
kStatus_ENET_Layer2UnInitialized Layer2 PTP buffer queue uninitialized.
kStatus_ENET_Layer2OverLarge Layer2 packet length over large.
kStatus_ENET_Layer2BufferFull Layer2 packet buffer full.
kStatus_ENET_Layer2TypeError Layer2 packet error type.
kStatus_ENET_PttringBufferFull PTP ring buffer full.
kStatus_ENET_PttringBufferEmpty PTP ring buffer empty.
kStatus_ENET_SMIUninitialized SMI uninitialized.
kStatus_ENET_SMIVisitTimeout SMI visit timeout.
kStatus_ENET_RxbdInvalid Receive buffer descriptor invalid.
kStatus_ENET_RxbdEmpty Receive buffer descriptor empty.
kStatus_ENET_RxbdTrunc Receive buffer descriptor truncate.
kStatus_ENET_RxbdError Receive buffer descriptor error.
kStatus_ENET_RxBdFull Receive buffer descriptor full.
kStatus_ENET_SmallRxBuffSize Receive buffer size is so small.
kStatus_ENET_NoEnoughRxBuffers Small receive buffer size.
kStatus_ENET_LargeBufferFull Receive large buffer full.
kStatus_ENET_TxLarge Transmit large packet.
kStatus_ENET_Txbdfull Transmit buffer descriptor full.
kStatus_ENET_TxbdNull Transmit buffer descriptor Null.
kStatus_ENET_TxBufferNull Transmit data buffer Null.
kStatus_ENET_NoRxBufferLeft No more receive buffer left.
kStatus_ENET_UnknownCommand Invalid ENET PTP IOCTL command.
kStatus_ENET_TimeOut ENET Timeout.
kStatus_ENET_MulticastPointerNull Null multicast group pointer.

ENET HAL driver

kStatus_ENET_NoMulticastAddr No multicast group address.
kStatus_ENET_AlreadyAddedMulticast Have Already added to multicast group.
kStatus_ENET_PHYAutoDiscoverFail Failed to automatically discover PHY.

10.1.3.2 enum enet_rx_bd_control_status_t

Enumerator

kEnetRxBdEmpty Empty bit.
kEnetRxBdRxSoftOwner1 Receive software owner.
kEnetRxBdWrap Update buffer descriptor.
kEnetRxBdRxSoftOwner2 Receive software owner.
kEnetRxBdLast Last BD in the frame.
kEnetRxBdMiss Receive for promiscuous mode.
kEnetRxBdBroadCast Broadcast.
kEnetRxBdMultiCast Multicast.
kEnetRxBdLengthViolation Receive length violation.
kEnetRxBdNoOctet Receive non-octet aligned frame.
kEnetRxBdCrc Receive CRC error.
kEnetRxBdOverRun Receive FIFO overrun.
kEnetRxBdTrunc Frame is truncated.

10.1.3.3 enum enet_rx_bd_control_extend0_t

Enumerator

kEnetRxBdIpv4 Ipv4 frame.
kEnetRxBdIpv6 Ipv6 frame.
kEnetRxBdVlan VLAN.
kEnetRxBdProtocolChecksumErr Protocol checksum error.
kEnetRxBdIpHeaderChecksumErr IP header checksum error.

10.1.3.4 enum enet_rx_bd_control_extend1_t

Enumerator

kEnetRxBdIntrrupt BD interrupt.
kEnetRxBdUnicast Unicast frame.
kEnetRxBdCollision BD collision.
kEnetRxBdPhyErr PHY error.
kEnetRxBdMacErr Mac error.

10.1.3.5 enum enet_tx_bd_control_status_t

Enumerator

- kEnetTxBdReady* Ready bit.
- kEnetTxBdTxSoftOwner1* Transmit software owner.
- kEnetTxBdWrap* Wrap buffer descriptor.
- kEnetTxBdTxSoftOwner2* Transmit software owner.
- kEnetTxBdLast* Last BD in the frame.
- kEnetTxBdTransmitCrc* Receive for transmit CRC.

10.1.3.6 enum enet_tx_bd_control_extend0_t

Enumerator

- kEnetTxBdTxErr* Transmit error.
- kEnetTxBdTxUnderFlowErr* Underflow error.
- kEnetTxBdExcessCollisionErr* Excess collision error.
- kEnetTxBdTxFrameErr* Frame error.
- kEnetTxBdLateCollisionErr* Late collision error.
- kEnetTxBdOverFlowErr* Overflow error.
- kEnetTxTimestampErr* Timestamp error.

10.1.3.7 enum enet_tx_bd_control_extend1_t

Enumerator

- kEnetTxBdTxInterrupt* Transmit interrupt.
- kEnetTxBdTimeStamp* Transmit timestamp flag.

10.1.3.8 enum enet_constant_parameter_t

Enumerator

- kEnetMacAddrLen* ENET mac address length.
- kEnetHashValMask* ENET hash value mask.
- kEnetMinBuffSize* ENET minimum buffer size.
- kEnetMaxTimeout* ENET timeout.
- kEnetMdcFreq* MDC frequency.

ENET HAL driver

10.1.3.9 enum enet_fifo_configure_t

Enumerator

kEnetMinTxFifoAlmostFull ENET minimum transmit fifo almost full value.

kEnetMinFifoAlmostEmpty ENET minimum FIFO almost empty value.

kEnetDefaultTxFifoAlmostFull ENET default tranmit fifo almost full value.

10.1.3.10 enum enet_mac_operate_mode_t

Enumerator

kEnetMacNormalMode Normal operationg mode for ENET MAC.

kEnetMacSleepMode Sleep mode for ENET MAC.

10.1.3.11 enum enet_config_rmii_mode_t

Enumerator

kEnetCfgMii MII mode for data interface.

kEnetCfgRmii RMII mode for data interface.

10.1.3.12 enum enet_config_speed_t

Enumerator

kEnetCfgSpeed100M Speed 100 M mode.

kEnetCfgSpeed10M Speed 10 M mode.

10.1.3.13 enum enet_config_duplex_t

Enumerator

kEnetCfgHalfDuplex Half duplex mode.

kEnetCfgFullDuplex Full duplex mode.

10.1.3.14 enum enet_mii_write_t

Enumerator

kEnetWriteNoCompliant Write frame operation, but not MII compliant.

kEnetWriteValidFrame Write frame operation for a valid MII management frame.

10.1.3.15 enum enet_mii_read_t

Enumerator

kEnetReadValidFrame Read frame operation for a valid MII management frame.

kEnetReadNoCompliant Read frame operation, but not MII compliant.

10.1.3.16 enum enet_mdio_holdon_clkcycle_t

Enumerator

kEnetMdioHoldOneClkCycle MDIO output hold on one clock cycle.

kEnetMdioHoldTwoClkCycle MDIO output hold on two clock cycles.

kEnetMdioHoldThreeClkCycle MDIO output hold on three clock cycles.

kEnetMdioHoldFourClkCycle MDIO output hold on four clock cycles.

kEnetMdioHoldFiveClkCycle MDIO output hold on five clock cycles.

kEnetMdioHoldSixClkCycle MDIO output hold on six clock cycles.

kEnetMdioHoldSevenClkCycle MDIO output hold seven two clock cycles.

kEnetMdioHoldEightClkCycle MDIO output hold on eight clock cycles.

10.1.3.17 enum enet_special_address_filter_t

Enumerator

kEnetSpecialAddressInit Initializes the special address filter.

kEnetSpecialAddressEnable Enables the special address filter.

kEnetSpecialAddressDisable Disables the special address filter.

10.1.3.18 enum enet_timer_channel_t

Enumerator

kEnetTimerChannel1 1588 timer Channel 1

kEnetTimerChannel2 1588 timer Channel 2

kEnetTimerChannel3 1588 timer Channel 3

kEnetTimerChannel4 1588 timer Channel 4

10.1.3.19 enum enet_timer_channel_mode_t

Enumerator

kEnetChannelDisable Disable timer channel.

kEnetChannelRisingCapture Input capture on rising edge.

ENET HAL driver

kNetChannelFallingCapture Input capture on falling edge.
kNetChannelBothCapture Input capture on both edges.
kNetChannelSoftCompare Output compare software only.
kNetChannelToggleCompare Toggle output on compare.
kNetChannelClearCompare Clear output on compare.
kNetChannelSetCompare Set output on compare.
kNetChannelClearCompareSetOverflow Clear output on compare, set output on overflow.
kNetChannelSetCompareClearOverflow Set output on compare, clear output on overflow.
kNetChannelPulseLowOnCompare Pulse output low on compare for one 1588 clock cycle.
kNetChannelPulseHighOnCompare Pulse output high on compare for one 1588 clock cycle.

10.1.3.20 enum enet_interrupt_request_t

Enumerator

kNetBabrInterrupt Babbling receive error interrupt source.
kNetBabiInterrupt Babbling transmit error interrupt source.
kNetGraceStopInterrupt Graceful stop complete interrupt source.
kNetTxFrameInterrupt TX FRAME interrupt source.
kNetTxByteInterrupt TX BYTE interrupt source.
kNetRxFrameInterrupt RX FRAME interrupt source.
kNetRxByteInterrupt RX BYTE interrupt source.
kNetMiiInterrupt MII interrupt source.
kNetEBusERInterrupt Ethernet bus error interrupt source.
kNetLateCollisionInterrupt Late collision interrupt source.
kNetRetryLimitInterrupt Collision Retry Limit interrupt source.
kNetUnderrunInterrupt Transmit FIFO underrun interrupt source.
kNetPayloadRxInterrupt Payload Receive interrupt source.
kNetWakeUpInterrupt WAKEUP interrupt source.
kNetTsAvailInterrupt TS AVAIL interrupt source.
kNetTsTimerInterrupt TS WRAP interrupt source.
kNetAllInterrupt All interrupt.

10.1.3.21 enum enet_irq_number_t

Enumerator

kNetTsTimerNumber ENET ts_timer irq number.
kNetReceiveNumber ENET receive irq number.
kNetTransmitNumber ENET transmit irq number.
kNetMiiErrorNumber ENET mii error irq number.

10.1.3.22 enum enet_frame_max_t

Enumerator

- kEnetMaxFrameSize* Maximum frame size.
- kEnetMaxFrameVlanSize* Maximum VLAN frame size.
- kEnetMaxFrameDataSize* Maximum frame data size.
- kEnetDefaultIpg* Default Truncate length. ENET default transmit inter packet gap
- kEnetMaxFrameBdNumbers* Maximum buffer descriptor numbers of a frame.
- kEnetFrameFcsLen* FCS length.
- kEnetEthernetHeadLen* Ethernet Frame header length.
- kEnetEthernetVlanHeadLen* Ethernet Vlan frame header length.

10.1.3.23 enum enet_mac_control_flag_t

Enumerator

- kEnetStopModeEnable* ENET Stop mode enable.
- kEnetPayloadlenCheckEnable* Enable MAC to enter hardware freeze when enter Debug mode. ENET receive payload length check Enable
- kEnetRxFlowControlEnable* ENET flow control, receiver detects PAUSE frames and stops transmitting data when a PAUSE frame is detected.
- kEnetRxCrcFwdEnable* Received frame crc is stripped from the frame.
- kEnetRxPauseFwdEnable* Pause frames are forwarded to the user application.
- kEnetRxPadRemoveEnable* Padding is removed from received frames.
- kEnetRxBcRejectEnable* Broadcast frame reject.
- kEnetRxPromiscuousEnable* Promiscuous mode enabled.
- kEnetTxCrcFwdEnable* Enable transmit frame with the crc from application.
- kEnetTxCrcBdEnable* When Tx CRC FWD disable, Tx buffer descriptor enable Transmit CRC.
- kEnetMacAddrInsert* Enable MAC address insert.
- kEnetTxAccelEnable* Transmit accelerator enable.
- kEnetRxAccelEnable* Transmit accelerator enable.
- kEnetStoreAndFwdEnable* Switcher to enable store and forward.
- kEnetMacMibEnable* Disable MIB module.
- kEnetSMIPreambleDisable* Enable SMI preamble.
- kEnetVlanTagEnabled* Enable Vlan Tag.
- kEnetMacEnhancedEnable* Enable enhanced MAC feature (1588 feature/enhanced buff descriptor)

10.1.4 Function Documentation

10.1.4.1 uint32_t ENET_HAL_Init (uint32_t baseAddr)

ENET HAL driver

Parameters

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

Returns

The status of the initialize operation.

- kStatus_ENET_InitTimeout initialize failure for timeout.
- kStatus_ENET_Success initialize success.

10.1.4.2 void ENET_HAL_SetMac (*uint32_t baseAddr*, *const enet_mac_config_t * macCfgPtr*, *uint32_t sysClk*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>macCfgPtr</i>	The mac configure structure.
<i>sysClk</i>	The system clock for ENET module.

10.1.4.3 void ENET_HAL_SetTxBuffDescriptors (*uint32_t baseAddr*, *volatile enet_bd_struct_t * txBds*, *uint8_t * txBuffer*, *uint32_t txBdNumber*, *uint32_t txBuffSizeAlign*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>txBds</i>	The start address of ENET transmit buffer descriptors. This address must always be evenly divisible by 16.
<i>txBuffer</i>	The transmit data buffer start address. This address must always be evenly divisible by 16.
<i>txBdNumber</i>	The transmit buffer descriptor numbers.
<i>txBuffSizeAlign</i>	The aligned transmit buffer size.

10.1.4.4 void ENET_HAL_SetRxBuffDescriptors (*uint32_t baseAddr*, *volatile enet_bd_struct_t * rxBds*, *uint8_t * rxBuffer*, *uint32_t rxBdNumber*, *uint32_t rxBuffSizeAlign*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>rxBds</i>	The start address of ENET receive buffer descriptors. This address must always be evenly divisible by 16.
<i>rxBuffer</i>	The receive data buffer start address. This address must always be evenly divisible by 16.
<i>rxBdNumber</i>	The receive buffer descriptor numbers.
<i>rxBuffSizeAlign</i>	The aligned receive transmit buffer size.

10.1.4.5 void ENET_HAL_SetFifo (uint32_t *baseAddr*, const enet_mac_config_t * *macCfgPtr*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>macCfgPtr</i>	The ENET MAC configuration structure.

10.1.4.6 void ENET_HAL_GetMibRxStat (uint32_t *baseAddr*, enet_mib_rx_stat_t * *rxStat*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>rxStat</i>	The received statistics from MIB.

10.1.4.7 void ENET_HAL_GetMibTxStat (uint32_t *baseAddr*, enet_mib_tx_stat_t * *txStat*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>txStat</i>	The received statistics from MIB.

ENET HAL driver

10.1.4.8 static void ENET_HAL_SetStopCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This controls device behavior in doze mode.

In doze mode, all clocks of the ENET assembly are disabled, except the RMII/MII clock. Doze mode is similar to a conditional stop mode entry for the ENET assembly depending on the stop enable signal control enable/disable.

Parameters

<i>baseAddr</i>	The ENET peripheral base address..
<i>enable</i>	The switch to enable/disable stop mode. <ul style="list-style-type: none">• true to enable the stop mode.• false to disable the stop mode.

10.1.4.9 static void ENET_HAL_SetDebugCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Enabling the debug mode enables Mac to enter the hardware freeze when the device enters debug mode. Disabling the debug mode enables Mac to continue operation in debug mode.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable the sleep mode. <ul style="list-style-type: none">• true MAC enter hardware freeze mode in debug mode.• false MAC continues operation in debug mode.

10.1.4.10 void ENET_HAL_SetMacMode (uint32_t *baseAddr*, enet_mac_operate_mode_t *mode*)

Enabling the sleep mode disables the normal operating mode. When enabling the sleep mode, the magic packet detection is also enabled so that a remote agent can wake up the node.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>mode</i>	The normal operating mode or sleep mode.

10.1.4.11 void ENET_HAL_SetMacAddr (*uint32_t baseAddr, uint8_t * hwAddr*)

This interface sets the six-byte Mac address of the ENET interface.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>hwAddr</i>	The pointer to the array of the six-bytes Mac address. The six-bytes mac address is used by ENET MAC to filtering the incoming Ethernet frames.

10.1.4.12 static void ENET_HAL_SetMacAddrInsertCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This interface enables the six-byte Mac address modification on transmit. If this is enabled, the MAC overwrites the source Mac address with the given Mac address through the ENET_HAL_SetMacAddr.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable the Mac address modification on transmit. <ul style="list-style-type: none"> • true enable Mac address modification on transmit. • false disable Mac address modification on transmit.

10.1.4.13 void ENET_HAL_SetMulticastAddrHash (*uint32_t baseAddr, uint32_t crcValue, enet_special_address_filter_t mode*)

This interface is used to add the ENET device to a multicast group address. After joining the group, Mac receives all frames with the group Mac address.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>crcValue</i>	The CRC value of the multicast group address.
<i>mode</i>	The operation for init/enable/disable the specified hardware address.

ENET HAL driver

10.1.4.14 void ENET_HAL_SetUnicastAddrHash (*uint32_t baseAddr, uint32_t crcValue, enet_special_address_filter_t mode*)

This interface is used to add an individual address to the hardware address filter. Mac receives all frames with the individual address as a destination address.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>crcValue</i>	The CRC value of the special address.
<i>mode</i>	The operation for init/enable/disable the specified hardware address.

10.1.4.15 static void ENET_HAL_SetTxcrcFwdCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

If transmitting the CRC forward is enabled, the transmitter does not append a CRC to transmitted frames, because it is expecting a frame with the CRC from the application. If transmitting the CRC forward is disabled, the transmit buffer descriptor controls whether the frame has a CRC from the application.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	<p>The switch to enable/disable forwarding frame from application with CRC for transmitted frames.</p> <ul style="list-style-type: none"> • True the transmitter forwarding the frames with CRC from the application. so the transmitter doesn't append any CRC to transmitted frames. • False the transmit buffer descriptor controls whether the frame has a CRC from the application.

10.1.4.16 static void ENET_HAL_SetRxcrcFwdCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function decides whether the CRC field of the received frame is transmitted or stripped. Enabling this feature strips the CRC field from the frame. If padding remove is enabled, this feature is ignored and the CRC field is checked and always terminated and removed. Note that if the padding is enabled, the CRC field is checked and always terminated and removed.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	<p>The switch to enable/disable transmit the receive CRC.</p> <ul style="list-style-type: none"> • True to transmit the received CRC. • False to strip the received CRC.

ENET HAL driver

**10.1.4.17 static void ENET_HAL_SetPauseFwdCmd (uint32_t *baseAddr*, bool *enable*)
[inline], [static]**

This is used to decide whether pause frames are forwarded or discarded.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable forward PAUSE frames <ul style="list-style-type: none"> • True to forward PAUSE frames. • False to terminate and discard PAUSE frames.

10.1.4.18 static void ENET_HAL_SetPadRemoveCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Enabling the frame padding remove removes the padding from the received frames.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable remove padding <ul style="list-style-type: none"> • True to remove padding from frames. • False to disable padding remove.

10.1.4.19 static void ENET_HAL_SetFlowControlCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

If the flow control is enabled, the receive detects paused frames. Upon pause frame detection, the transmitter stops transmitting data frames for a given duration.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable flow control. <ul style="list-style-type: none"> • True to enable the flow control. • False to disable the flow control.

10.1.4.20 static void ENET_HAL_SetBroadcastRejectCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

If the broadcast frame reject is enabled, frames with destination address equal to 0xffff_ffff are rejected unless the promiscuous mode is open.

ENET HAL driver

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable reject broadcast frames. <ul style="list-style-type: none">• True to reject broadcast frames.• False to accept broadcast frames.

10.1.4.21 static void ENET_HAL_SetPayloadCheckCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

If the length/type is less than 0x600, when enable payload length check is enabled, the core checks the frame's payload length. If the length/type is greater than or equal to 0x600, Mac interprets the field as a type and no payload length check is performed.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable payload length check. <ul style="list-style-type: none">• True to enable payload length check.• False to disable payload length check.

10.1.4.22 static void ENET_HAL_SetGraceTxStopCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

When this field is set, Mac stops transmission after a currently transmitted frame is complete.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable graceful transmit stop <ul style="list-style-type: none">• True to enable graceful transmit stop.• False to disable graceful transmit stop.

10.1.4.23 static void ENET_HAL_SetPauseDuration (uint32_t *baseAddr*, uint32_t *pauseDuration*) [inline], [static]

This function sets the pause duration used in a transmission of a PAUSE frame. When another node detects a PAUSE frame, that node pauses transmission for the pause duration.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>pauseDuration</i>	The PAUSE duration for the transmitted PAUSE frame the maximum pause duration is 0xFFFF.

10.1.4.24 static void ENET_HAL_SetTxPauseCmd (*uint32_t baseAddr, bool enable*) [**inline**], [**static**]

This function enables pausing the frame transmission. When this is set, with transmission of data frames stopped, Mac transmits a Mac control pause frame. Next, Mac clears and resumes transmitting data frames.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable transmit pause frame.

10.1.4.25 static bool ENET_HAL_GetTxPause (*uint32_t baseAddr*) [**inline**], [**static**]

This function gets the transmitted pause frame status.

Parameters

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

Returns

The status of the received flow control frames.

- True if the MAC is transmitting a MAC control PAUSE frame.
- False if No PAUSE frame transmit.

10.1.4.26 static bool ENET_HAL_GetRxPause (*uint32_t baseAddr*) [**inline**], [**static**]

This function gets the received pause frame status.

ENET HAL driver

Parameters

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

Returns

The status of the received flow control frames

- True if the flow control pause frame is received and the transmitter pauses for the duration defined in this pause frame.
- False if there is no flow control frame received or the pause duration is complete.

10.1.4.27 static void ENET_HAL_SetTxInterPacketGap (*uint32_t baseAddr, uint32_t ipgValue*) [inline], [static]

This function indicates the inter-packet gap, in bytes, between transmitted frames. Valid values range from 8 to 27. If value is less than 8, the IPG is 8. If value is greater than 27, the IPG is 27.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>ipgValue</i>	The Inter-Packet-Gap for transmitted frames The default value is 12, the maximum value set to ipg is 0x1F.

10.1.4.28 static void ENET_HAL_SetTruncLen (*uint32_t baseAddr, uint32_t length*) [inline], [static]

This function indicates the value a receive frame is truncated, if it is greater than this value. The frame truncation length must be greater than or equal to the receive maximum frame length.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>length</i>	The truncation length. The maximum value is 0x3FFF The default truncation length is 2047(0x7FF).

10.1.4.29 static void ENET_HAL_SetRxMaxFrameLen (*uint32_t baseAddr, uint32_t maxFrameSize*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>maxFrameSize</i>	The maximum receive frame size, the reset value is 1518 or 1522 if the VLAN tags are supported. The length is measured starting at DA and including the CRC.

10.1.4.30 static void ENET_HAL_SetRxMaxBuffSize (*uint32_t baseAddr, uint32_t maxBufferSize*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>maxBufferSize</i>	The maximum receive buffer size, which should not be smaller than 256 It should be evenly divisible by 16 and the maximum receive size should not be larger than 0x3ff0.

10.1.4.31 void ENET_HAL_SetTxFifo (*uint32_t baseAddr, enet_config_tx_fifo_t * thresholdCfg*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>thresholdCfg</i>	The FIFO threshold configuration

10.1.4.32 void ENET_HAL_SetRxFifo (*uint32_t baseAddr, enet_config_rx_fifo_t * thresholdCfg*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>thresholdCfg</i>	The FIFO threshold configuration

10.1.4.33 static void ENET_HAL_SetRxBuffDescripAddr (*uint32_t baseAddr, uint32_t rxBdAddr*) [inline], [static]

This interface provides the beginning of the receive buffer descriptor queue in the external memory. The rxBdAddr is recommended to be 128-bit aligned, must be evenly divisible by 16.

ENET HAL driver

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>txBdAddr</i>	The start address of receive buffer descriptor. This address must always be evenly divisible by 16.

10.1.4.34 static void ENET_HAL_SetTxBuffDescripAddr (*uint32_t baseAddr, uint32_t txBdAddr*) [inline], [static]

This interface provides the beginning of the transmit buffer descriptor queue in the external memory. The txBdAddr is recommended to be 128-bit aligned, must be evenly divisible by 16.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>txBdAddr</i>	The start address of transmit buffer descriptors This address must always be evenly divisible by 16.

10.1.4.35 void ENET_HAL_InitRxBuffDescriptors (*volatile enet_bd_struct_t * rxBds, uint8_t * rxBuff, uint32_t rxbdNum, uint32_t rxBuffSizeAlign*)

To make sure the uDMA will do the right data transfer after you activate with wrap flag and all the buffer descriptors should be initialized with an empty bit.

Parameters

<i>rxBds</i>	The current receive buffer descriptor.
<i>rxBuff</i>	The data buffer on receive buffer descriptor. This address must always be evenly divisible by 16.
<i>rxbdNum</i>	The number of the receive buffer descriptors.
<i>rxBuffSizeAlign</i>	The aligned receive buffer size.

10.1.4.36 void ENET_HAL_InitTxBuffDescriptors (*volatile enet_bd_struct_t * txBds, uint8_t * txBuff, uint32_t txbdNum, uint32_t txBuffSizeAlign*)

To make sure the uDMA will do the right data transfer after you active with wrap flag.

Parameters

<i>txBds</i>	The current transmit buffer descriptor.
<i>txBuff</i>	The data buffer on transmit buffer descriptor.
<i>txbdNum</i>	The number of transmit buffer descriptors.
<i>txBuffSizeAlign</i>	The aligned transmit buffer size.

10.1.4.37 void ENET_HAL_UpdateRxBuffDescriptor (volatile enet_bd_struct_t * *rxBds*, uint8_t * *data*, bool *isbufferUpdate*)

This interface mainly clears the status region and updates the received buffer descriptor to ensure that the BD is correctly used.

Parameters

<i>rxBds</i>	The current receive buffer descriptor.
<i>data</i>	The data buffer address. This address must be divided by 16 if the <i>isbufferUpdate</i> is set.
<i>isbufferUpdate</i>	The data buffer update flag. When you want to update the data buffer of the buffer descriptor ensure that this flag is set.

10.1.4.38 void ENET_HAL_UpdateTxBuffDescriptor (volatile enet_bd_struct_t * *txBds*, uint16_t *length*, bool *isTxtsCfged*, bool *isTxCrcEnable*, bool *isLastOne*)

This interface mainly clears the status region and updates the transmit buffer descriptor to ensure tat the BD is correctly used again. You should set the *isTxtsCfged* when the transmit timestamp feature is required.

ENET HAL driver

Parameters

<i>txBds</i>	The current transmit buffer descriptor.
<i>length</i>	The data length on buffer descriptor.
<i>isTxtsCfged</i>	The timestamp configure flag. The timestamp is added to the transmit buffer descriptor when this flag is set.
<i>isTxCrcEnable</i>	<p>The flag to transmit CRC sequence after the data byte.</p> <ul style="list-style-type: none">• True the transmit controller transmits the CRC sequence after the data byte. if the transmit CRC forward from application is disabled this flag should be set to add the CRC sequence.• False the transmit buffer descriptor does not transmit the CRC sequence after the data byte. if the transmit CRC forward from application.
<i>isLastOne</i>	<p>The last BD flag in a frame.</p> <ul style="list-style-type: none">• True the last BD in a frame.• False not the last BD in a frame.

10.1.4.39 static void ENET_HAL_ClearTxBuffDescriptor (volatile enet_bd_struct_t * *curBd*) [inline], [static]

Clears the data, length, control, and status region of the transmit buffer descriptor.

Parameters

<i>curBd</i>	The current buffer descriptor.
--------------	--------------------------------

10.1.4.40 static uint16_t ENET_HAL_GetRxBuffDescripControl (volatile enet_bd_struct_t * *curBd*) [inline], [static]

This interface can get the whole control and status region of the receive buffer descriptor. The enet_rx_bd_control_status_t enum type definition should be used if you want to get each status bit of the control and status region.

Parameters

<i>curBd</i>	The current receive buffer descriptor.
--------------	--

Returns

The control and status data on buffer descriptors.

10.1.4.41 static uint16_t ENET_HAL_GetTxBuffDescripControl (volatile enet_bd_struct_t * *curBd*) [inline], [static]

This interface can get the whole control and status region of the transmit buffer descriptor. The enet_tx_bd_control_status_t enum type definition should be used if you want to get each status bit of the control and status region.

Parameters

<i>curBd</i>	The current transmit buffer descriptor.
--------------	---

Returns

The extended control region of transmit buffer descriptor.

10.1.4.42 static uint16_t ENET_HAL_GetRxbuffDescripExtControlOne (volatile enet_bd_struct_t * *curBd*) [inline], [static]

This interface can get the whole control and status region of the receive buffer descriptor. The enet_rx_bd_control_extend0_t enum type definition should be used if you want to get each status bit of the control and status region.

Parameters

<i>curBd</i>	The current receive buffer descriptor.
--------------	--

Returns

The extended control region0 data of receive buffer descriptor.

10.1.4.43 static uint16_t ENET_HAL_GetRxbuffDescripExtControlTwo (volatile enet_bd_struct_t * *curBd*) [inline], [static]

This interface can get the whole control and status region of the receive buffer descriptor. The enet_rx_bd_control_extend1_t enum type definition should be used if you want to get each status bit of the control and status region.

Parameters

ENET HAL driver

<i>curBd</i>	The current receive buffer descriptor.
--------------	--

Returns

The extended control region1 data of receive buffer descriptor.

10.1.4.44 static uint16_t ENET_HAL_GetTxBuffDescripExtControl (volatile enet_bd_struct_t * *curBd*) [inline], [static]

This interface can get the whole control and status region of the transmit buffer descriptor. The enet_tx_bd_control_extend0_t enum type definition should be used if you want to get each status bit of the control and status region.

Parameters

<i>curBd</i>	The current transmit buffer descriptor.
--------------	---

Returns

The extended control data.

10.1.4.45 static bool ENET_HAL_GetTxBuffDescripTsFlag (volatile enet_bd_struct_t * *curBd*) [inline], [static]

Parameters

<i>curBd</i>	The ENET transmit buffer descriptor.
--------------	--------------------------------------

Returns

true if timestamp region is set else false.

10.1.4.46 uint16_t ENET_HAL_GetBuffDescripLen (volatile enet_bd_struct_t * *curBd*)

Parameters

<i>curBd</i>	The current buffer descriptor.
--------------	--------------------------------

Returns

The data length of the buffer descriptor.

10.1.4.47 `uint8_t* ENET_HAL_GetBuffDescripData (volatile enet_bd_struct_t * curBd)`

ENET HAL driver

Parameters

<i>curBd</i>	The current buffer descriptor.
--------------	--------------------------------

Returns

The buffer address of the buffer descriptor.

10.1.4.48 **uint32_t ENET_HAL_GetBuffDescripTs (volatile enet_bd_struct_t * *curBd*)**

Parameters

<i>curBd</i>	The current buffer descriptor.
--------------	--------------------------------

Returns

The time stamp of the frame in the buffer descriptor. Notice that the frame timestamp is only set in the last buffer descriptor of the frame.

10.1.4.49 **static void ENET_HAL_SetRxBuffDescripActive (uint32_t *baseAddr*) [inline], [static]**

The buffer descriptor activation should be done after the ENET module is enabled. Otherwise, the activation fails.

Parameters

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

10.1.4.50 **static void ENET_HAL_SetTxBuffDescripActive (uint32_t *baseAddr*) [inline], [static]**

The buffer descriptor activation should be done after the ENET module is enabled. Otherwise, the activation fails.

Parameters

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

10.1.4.51 void ENET_HAL_SetRMIIMode (uint32_t *baseAddr*, enet_config_rmii_t * *rmiiCfgPtr*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>rmiiCfgPtr</i>	The RMII/MII configuration structure pointer.

10.1.4.52 static void ENET_HAL_SetRMIISpeed (uint32_t *baseAddr*, enet_config_speed_t *speed*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>speed</i>	The RMII/MII speed.

10.1.4.53 static void ENET_HAL_SetSMI (uint32_t *baseAddr*, uint32_t *miiSpeed*, enet_mdio_holdon_clkcycle_t *clkCycle*, bool *isPreambleDisabled*) [inline], [static]

Sets the SMI(MDC/MDIO) between Mac and PHY. The miiSpeed is a value that controls the frequency of the MDC, relative to the internal module clock(InterClockSrc). A value of zero in this parameter turns the MDC off and leaves it in the low voltage state. Any non-zero value results in the MDC frequency MDC = InterClockSrc/((miiSpeed + 1)*2). So miiSpeed = InterClockSrc/(2*MDC) - 1. The Maximum MDC clock is 2.5MHZ(maximum). The recommended action is to round up and plus one to simplify: miiSpeed = InterClockSrc/(2*2.5MHZ).

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>miiSpeed</i>	The MII speed and it is ranged from 0~0x3F.
<i>clkCycle</i>	The hold on clock cycles for MDIO output.
<i>isPreambleDisabled</i>	The preamble disabled flag.

ENET HAL driver

10.1.4.54 static bool ENET_HAL_GetSMI(uint32_t *baseAddr*) [inline], [static]

This interface is usually called to check the SMI(serial Management interface) before the Mac writes or reads the PHY registers.

Parameters

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

Returns

The MII configuration status.

- true if the MII has been configured.
- false if the MII has not been configured.

10.1.4.55 static uint32_t ENET_HAL_GetSMIData (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The data read from PHY

10.1.4.56 void ENET_HAL_SetSMIRead (uint32_t *baseAddr*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_read_t *operation*)

Parameters

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

10.1.4.57 void ENET_HAL_SetSMIWrite (uint32_t *baseAddr*, uint32_t *phyAddr*, uint32_t *phyReg*, enet_mii_write_t *operation*, uint32_t *data*)

ENET HAL driver

Parameters

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

10.1.4.58 static void ENET_HAL_Enable(uint32_t *baseAddr*) [inline], [static]

Parameters

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

10.1.4.59 static void ENET_HAL_Disable(uint32_t *baseAddr*) [inline], [static]

Parameters

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

10.1.4.60 static void ENET_HAL_SetEnhancedMacCmd(uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The enable/disable switch to the enhanced functionality of the MAC(1588 feature).

10.1.4.61 void ENET_HAL_SetIntMode(uint32_t *baseAddr*, enet_interrupt_request_t *source*, bool *enable*)

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>source</i>	The interrupt sources.
<i>enable</i>	The interrupt enable switch.

10.1.4.62 static void ENET_HAL_ClearIntStatusFlag (uint32_t *baseAddr*, enet_interrupt_request_t *source*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>source</i>	The interrupt source to be cleared. enet_interrupt_request_t enum types is recommended as the interrupt source.

10.1.4.63 static bool ENET_HAL_GetIntStatusFlag (uint32_t *baseAddr*, enet_interrupt_request_t *source*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>source</i>	The interrupt sources. enet_interrupt_request_t enum types is recommended as the interrupt source.

Returns

- The event status of the interrupt source
- true if the interrupt event happened.
 - false if the interrupt event has not happened.

10.1.4.64 static void ENET_HAL_SetPromiscuousCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable the promiscuous mode.

ENET HAL driver

```
10.1.4.65 static void ENET_HAL_SetMibClearCmd ( uint32_t baseAddr, bool enable )  
[inline], [static]
```

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable the MIB counter.

**10.1.4.66 static void ENET_HAL_SetMibCmd (uint32_t *baseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The switch to enable/disable the MIB block. <ul style="list-style-type: none"> • True to enable MIB block. • False to disable MIB block.

**10.1.4.67 static bool ENET_HAL_GetMibStatus (uint32_t *baseAddr*) [inline],
[static]**

Parameters

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

Returns

true if in MIB idle and MIB is not updating else false.

**10.1.4.68 void ENET_HAL_SetTxAccelerator (uint32_t *baseAddr*,
enet_config_tx_accelerator_t * *txCfgPtr*)**

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>txCfgPtr</i>	The transmit accelerator configuration.

**10.1.4.69 void ENET_HAL_SetRxAccelerator (uint32_t *baseAddr*,
enet_config_rx_accelerator_t * *rxCfgPtr*)**

ENET HAL driver

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>rxCfgPtr</i>	The receive accelerator configuration.

10.1.4.70 void ENET_HAL_Set1588TimerStart (uint32_t *baseAddr*, enet_config_ptp_timer_t * *ptpCfgPtr*)

This interface configures the 1588 timer and starts the 1588 timer. After the timer starts the 1588 timer starts incrementing.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>ptpCfgPtr</i>	The 1588 timer configuration structure pointer.

10.1.4.71 void ENET_HAL_Set1588TimerChnCmp (uint32_t *baseAddr*, enet_timer_channel_t *channel*, uint32_t *cmpValOld*, uint32_t *cmpValNew*)

This interface configures the 1588 timer channel with the compare feature.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>channel</i>	The 1588 timer channel.
<i>cmpValOld</i>	The old compare value.
<i>cmpValNew</i>	The new compare value.

10.1.4.72 void ENET_HAL_Set1588Timer (uint32_t *baseAddr*, enet_config_ptp_timer_t * *ptpCfgPtr*)

This interface initializes the 1588 context structure. Initialize 1588 parameters according to the user configuration structure.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>ptpCfgPtr</i>	The 1588 timer configuration.

10.1.4.73 static void ENET_HAL_Set1588TimerCmd (uint32_t *baseAddr*, uint32_t *enable*) [inline], [static]

Enable the PTP timer will starts the timer. Disable the timer will stop timer at the current value.

ENET HAL driver

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>enable</i>	The 1588 timer Enable switch <ul style="list-style-type: none">• True enabled the 1588 PTP timer.• False disable or stop the 1588 PTP timer.

10.1.4.74 static void ENET_HAL_Set1588TimerRestart (uint32_t *baseAddr*) [inline], [static]

Restarting the PTP timer clears all PTP-timer counters to zero.

Parameters

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

10.1.4.75 static void ENET_HAL_Set1588TimerAdjust (uint32_t *baseAddr*, uint32_t *increaseCorrection*, uint32_t *periodCorrection*) [inline], [static]

Adjust the 1588 timer according to the increase and correction period of the configured correction.

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>increase-Correction</i>	The increase correction for 1588 timer.
<i>period-Correction</i>	The period correction for 1588 timer.

10.1.4.76 static void ENET_HAL_Set1588TimerCapture (uint32_t *baseAddr*) [inline], [static]

This is used before reading the current time register. After set timer capture, please wait for about 1us before read the captured timer.

Parameters

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

10.1.4.77 static void ENET_HAL_Set1588TimerNewTime (*uint32_t baseAddr, uint32_t nanSecond*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>nanSecond</i>	The nanosecond set to 1588 timer.

10.1.4.78 static uint32_t ENET_HAL_Get1588TimerCurrentTime (*uint32_t baseAddr*) [inline], [static]

Parameters

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

Returns

the current time from 1588 timer.

10.1.4.79 static void ENET_HAL_Set1588TimerChnMode (*uint32_t baseAddr, enet_timer_channel_t channel, enet_timer_channel_mode_t mode*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>channel</i>	The 1588 timer channel number.
<i>mode</i>	Compare or capture mode for the four-channel 1588 timer channel.

10.1.4.80 static void ENET_HAL_Set1588TimerChnInt (*uint32_t baseAddr, enet_timer_channel_t channel, bool enable*) [inline], [static]

ENET HAL driver

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>channel</i>	The 1588 timer channel number.
<i>enable</i>	The switch to enable or disable interrupt.

10.1.4.81 static void ENET_HAL_Set1588TimerChnCmpVal (uint32_t *baseAddr*, enet_timer_channel_t *channel*, uint32_t *compareValue*) [inline], [static]

Parameters

<i>baseAddr</i>	The ENET peripheral base address.
<i>channel</i>	The 1588 timer channel number.
<i>compareValue</i>	Compare value for 1588 timer channel.

10.1.4.82 static bool ENET_HAL_Get1588TimerChnStatus (uint32_t *baseAddr*, enet_timer_channel_t *channel*) [inline], [static]

Parameters

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

Returns

Compare or capture operation status

- True if the compare or capture has occurred.
- False if the compare or capture has not occurred.

10.1.4.83 static void ENET_HAL_Clear1588TimerChnFlag (uint32_t *baseAddr*, enet_timer_channel_t *channel*) [inline], [static]

Parameters

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

10.1.4.84 static uint32_t ENET_HAL_GetTxTs (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The timestamp of the last transmitted frame.

10.1.4.85 static uint16_t ENET_HAL_GetTxPackets (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packet count.

10.1.4.86 static uint16_t ENET_HAL_GetTxBroadCastPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit broadcast packet statistic.

10.1.4.87 static uint16_t ENET_HAL_GetTxMultiCastPacket (uint32_t *baseAddr*) [inline], [static]

ENET HAL driver

Parameters

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

Returns

The transmit multicast packet statistic.

10.1.4.88 static uint16_t ENET_HAL_GetTxCrcAlignErrorPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets with CRC/Align error.

10.1.4.89 static uint16_t ENET_HAL_GetTxUnderSizePacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets less than 64 bytes and good CRC.

10.1.4.90 static uint16_t ENET_HAL_GetTxFragPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets less than 64 bytes and bad CRC.

10.1.4.91 **static uint16_t ENET_HAL_GetTxOverSizePacket (uint32_t *baseAddr*)**
[**inline**], [**static**]

ENET HAL driver

Parameters

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

Returns

The transmit packets over size than MAX_FL and good CRC.

10.1.4.92 static uint16_t ENET_HAL_GetTxJabPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets over size than MAX_FL bytes and bad CRC.

10.1.4.93 static uint16_t ENET_HAL_GetTxCollisionPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit collision packets.

10.1.4.94 static uint16_t ENET_HAL_GetTxByte64Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit 64-byte packet.

10.1.4.95 **static uint16_t ENET_HAL_GetTxByte65to127Packet (uint32_t *baseAddr*)**
[**inline**], [**static**]

ENET HAL driver

Parameters

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

Returns

The transmit 65-byte to 127-byte packet statistic.

10.1.4.96 static uint16_t ENET_HAL_GetTxByte128to255Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets 128 byte to 255-byte.

10.1.4.97 static uint16_t ENET_HAL_GetTxByte256to511Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets 256 byte to 511-byte.

10.1.4.98 static uint16_t ENET_HAL_GetTxByte512to1023Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets 512 byte to 1023-byte.

10.1.4.99 **static uint16_t ENET_HAL_GetTxByte1024to2047Packet (uint32_t *baseAddr*)**
[**inline**], [**static**]

ENET HAL driver

Parameters

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

Returns

The transmit packets 1024 byte to 2047-byte.

10.1.4.100 static uint16_t ENET_HAL_GetTxOverByte2048Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit packets greater than 2048-bytes.

10.1.4.101 static uint32_t ENET_HAL_GetTxOctets (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The transmit Octets.

10.1.4.102 static uint16_t ENET_HAL_GetTxFramesOk (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The Frames transmitted well.

10.1.4.103 **static uint16_t ENET_HAL_GetTxFramesOneCollision (uint32_t *baseAddr*)**
[**inline**], [**static**]

ENET HAL driver

Parameters

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

Returns

The frames transmitted with single collision.

10.1.4.104 static uint16_t ENET_HAL_GetTxFramesMultiCollision (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The frames transmitted with multiple collision.

10.1.4.105 static uint16_t ENET_HAL_GetTxFrameCarrSenseError (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The frames transmitted with carrier sense error.

10.1.4.106 static uint16_t ENET_HAL_GetTxFramesDelay (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The frames transmitted after deferral delay

**10.1.4.107 static uint16_t ENET_HAL_GetTxFramesLateCollision (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The frames transmitted with late collision.

**10.1.4.108 static uint16_t ENET_HAL_GetTxFramesExcessiveCollision (uint32_t
baseAddr) [inline], [static]**

Parameters

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

Returns

The frames transmitted with excessive collisions.

**10.1.4.109 static uint16_t ENET_HAL_GetTxFramesMacError (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The Frames transmitted with the Tx FIFO underrun.

**10.1.4.110 static uint16_t ENET_HAL_GetTxFramesPause (uint32_t *baseAddr*)
[inline], [static]**

ENET HAL driver

Parameters

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

Returns

The transmitted flow control Pause Frames.

10.1.4.111 static uint32_t ENET_HAL_GetTxOctetFramesOk (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The octet count for frames transmitted without error.

10.1.4.112 static uint16_t ENET_HAL_GetRxPackets (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packet count.

10.1.4.113 static uint16_t ENET_HAL_GetRxBroadCastPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive broadcast packet count.

10.1.4.114 **static uint16_t ENET_HAL_GetRxMultiCastPacket (uint32_t *baseAddr*)**
[**inline**], [**static**]

ENET HAL driver

Parameters

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

Returns

The receive multicast packet count.

10.1.4.115 static uint16_t ENET_HAL_GetRxCrcAlignErrorPacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packets with CRC/Align error.

10.1.4.116 static uint16_t ENET_HAL_GetRxUnderSizePacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packets less than 64-byte and good CRC.

10.1.4.117 static uint16_t ENET_HAL_GetRxOverSizePacket (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packets greater than MAX_FL and good CRC.

10.1.4.118 static uint16_t ENET_HAL_GetRxFragPacket (uint32_t *baseAddr*)
[inline], [static]

ENET HAL driver

Parameters

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

Returns

The receive packets less than 64-byte and bad CRC.

**10.1.4.119 static uint16_t ENET_HAL_GetRxJabPacket (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The receive packets greater than MAX_FL and bad CRC.

**10.1.4.120 static uint16_t ENET_HAL_GetRxByte64Packet (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The receive packets with 64-byte.

**10.1.4.121 static uint16_t ENET_HAL_GetRxByte65to127Packet (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The receive packets with 65-byte to 127-byte.

10.1.4.122 static uint16_t ENET_HAL_GetRxByte128to255Packet (uint32_t *baseAddr*)
[inline], [static]

ENET HAL driver

Parameters

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

Returns

The receive packets with 128-byte to 255-byte.

10.1.4.123 static uint16_t ENET_HAL_GetRxByte256to511Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packets with 256-byte to 511-byte.

10.1.4.124 static uint16_t ENET_HAL_GetRxByte512to1023Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packets with 512-byte to 1023-byte.

10.1.4.125 static uint16_t ENET_HAL_GetRxByte1024to2047Packet (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The receive packets with 1024-byte to 2047-byte.

**10.1.4.126 static uint16_t ENET_HAL_GetRxOverByte2048Packet (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The receive packets greater than 2048.

**10.1.4.127 static uint32_t ENET_HAL_GetRxOctets (uint32_t *baseAddr*) [inline],
[static]**

Parameters

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

Returns

The receive octets.

**10.1.4.128 static uint16_t ENET_HAL_GetRxFramesDrop (uint32_t *baseAddr*)
[inline], [static]**

If a frame with invalid or missing SFD character is detected and has been dropped.

Parameters

ENET HAL driver

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

Returns

The receive Frames not counted correctly.

**10.1.4.129 static uint16_t ENET_HAL_GetRxFramesOk (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The Frames received OK.

**10.1.4.130 static uint16_t ENET_HAL_GetRxFramesCrcError (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The Frames received with CRC error.

**10.1.4.131 static uint16_t ENET_HAL_GetRxFramesAlignError (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The Frames received with Alignment error.

**10.1.4.132 static uint16_t ENET_HAL_GetRxFramesMacError (uint32_t *baseAddr*)
[inline], [static]**

Parameters

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

Returns

The FIFO overflow count.

10.1.4.133 static uint16_t ENET_HAL_GetRxFramesFlowControl (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The received flow control Pause frames.

10.1.4.134 static uint32_t ENET_HAL_GetRxOtetsFramesOk (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

The octet count for frames received without error.

ENET Peripheral Driver

10.2 ENET Peripheral Driver

This chapter describes the programming interface of the ENET Peripheral Driver.

Data Structures

- struct [enet_multicast_group_t](#)
Defines the multicast group structure for the ENET device. [More...](#)
- struct [enet_ethernet_header_t](#)
Defines the ENET header structure. [More...](#)
- struct [enet_8021vlan_header_t](#)
Defines the ENET VLAN frame header structure. [More...](#)
- struct [enet_buff_descrip_context_t](#)
Defines the structure for ENET buffer descriptors stats. [More...](#)
- struct [enet_stats_t](#)
Defines the ENET packets statistic structure. [More...](#)
- struct [enet_mac_packet_buffer_t](#)
Defines the ENET MAC packet buffer structure. [More...](#)
- struct [enet_buff_config_t](#)
Defines the receive buffer descriptor configure structure. [More...](#)
- struct [enet_dev_if_t](#)
Defines the ENET device data structure for the ENET. [More...](#)

Macros

- #define [ENET_RECEIVE_ALL_INTERRUPT](#) 1
Defines the approach: ENET interrupt handler do receive.
- #define [ENET_ENABLE_DETAIL_STATS](#) 0
Defines the statistic enable macro.
- #define [ENET_ORIGINAL_CRC32](#) 0xFFFFFFFFU
Defines the CRC-32 calculation constant.
- #define [ENET_CRC32_POLYNOMIC](#) 0xEDB88320U
CRC-32 Polynomic.

Enumerations

- enum [enet_crc_parameter_t](#) {
 kEnetCrcOffset = 8,
 kEnetCrcMask1 = 0x3F }
Defines the CRC data for a hash value calculation.
- enum [enet_protocol_type_t](#) {
 kEnetProtocol2ptpv2 = 0x88F7,
 kEnetProtocolIpv4 = 0x0800,
 kEnetProtocolIpv6 = 0x86dd,
 kEnetProtocol8021QVlan = 0x8100,
 kEnetPacketUdpVersion = 0x11,
 kEnetPacketIpv4Version = 0x4,

```
kEnetPacketIpv6Version = 0x6 }
```

Defines the ENET protocol type and main parameters.

ENET Driver

- `uint32_t ENET_DRV_Init (enet_dev_if_t *enetIfPtr, const enet_mac_config_t *macCfgPtr, enet_buff_config_t *buffCfgPtr)`

Initializes the ENET with the basic configuration.
- `uint32_t ENET_DRV_Deinit (enet_dev_if_t *enetIfPtr)`

Deinitializes the ENET device.
- `uint32_t ENET_DRV_UpdateRxBuffDescrip (enet_dev_if_t *enetIfPtr, bool isBuffUpdate)`

Updates the receive buffer descriptor.
- `uint32_t ENET_DRV_CleanupTxBuffDescrip (enet_dev_if_t *enetIfPtr)`

ENET transmit buffer descriptor cleanup.
- `volatile enet_bd_struct_t * ENET_DRV_IncrRxBuffDescripIndex (enet_dev_if_t *enetIfPtr, volatile enet_bd_struct_t *curBd)`

Increases the receive buffer descriptor to the next one.
- `volatile enet_bd_struct_t * ENET_DRV_IncrTxBuffDescripIndex (enet_dev_if_t *enetIfPtr, volatile enet_bd_struct_t *curBd)`

Increases the transmit buffer descriptor to the next one.
- `bool ENET_DRV_RxErrorStats (enet_dev_if_t *enetIfPtr, uint32_t data)`

Processes the ENET receive frame error statistics.
- `void ENET_DRV_TxErrorStats (enet_dev_if_t *enetIfPtr, volatile enet_bd_struct_t *curBd)`

Processes the ENET transmit frame statistics.
- `uint32_t ENET_DRV_ReceiveData (enet_dev_if_t *enetIfPtr)`

Receives ENET packets.
- `uint32_t ENET_DRV_SendData (enet_dev_if_t *enetIfPtr, uint32_t dataLen, uint32_t bdNumUsed)`

Transmits ENET packets.
- `void ENET_DRV_RxIRQHandler (uint32_t instance)`

The ENET receive interrupt handler.
- `void ENET_DRV_TxIRQHandler (uint32_t instance)`

The ENET transmit interrupt handler.
- `void ENET_DRV_CalculateCrc32 (uint8_t *address, uint32_t *crcValue)`

Calculates the CRC hash value.
- `uint32_t ENET_DRV_AddMulticastGroup (uint32_t instance, uint8_t *address, uint32_t *hash)`

Adds the ENET device to a multicast group.
- `uint32_t ENET_DRV_LeaveMulticastGroup (uint32_t instance, uint8_t *address)`

Moves the ENET device from a multicast group.
- `void enet_mac_enqueue_buffer (void **queue, void *buffer)`

ENET buffer enqueue.
- `void * enet_mac_dequeue_buffer (void **queue)`

ENET buffer dequeue.

10.2.0.135 ENET Driver

Overview

The ENET driver receives data from and transmits data to the wired network. The enhanced 1588 feature supports clock synchronization.

ENET device data structure

The ENET device data structure, `enet_dev_if_t`, includes configuration structure, application structure, and data context structure. This structure should be initialized before the ENET device initialization function. Then, the data structure is passed from the API layer to the device.

Configuration structure

The ENET device data structure uses the `enet_mac_config_t` and the `enet_phy_config_t` configuration structure for MAC and PHY configurations. This allows users to configure the most common settings of the ENET peripheral. The `enet_mac_config_t` mac configuration structure includes the Mac mode, receive maximum frame length, receive truncate length, pause duration for pause frame, MII/RMII mode configure, MAC controller configure, receive and transmit accelerator, receive and transmit FIFO, extended buffer to receive the buffer descriptor and update the data buffer, and the PTP slave mode. The `enet_phy_config_t` PHY configuration structure includes the PHY address and PHY loop mode, which needs to be configured. If the PHY address is unknown, use the `isAutodiscoverEnabled` flag to find the PHY address. Note:

1. The recommended maximum frame length is 1518 or 1522(VLAN).
2. The recommended receive and transmit buffer size is the maximum frame size 1518 or 1522(VLA-N).
3. The extended buffer queue in the structure is used to receive the buffer descriptor and update the data buffer. This buffer queue is required when receiving API by using the polling mode.

Buffer data structure

The ENET device data structure uses the `enet_buff_config_t` to configure the receive and transmit buffer descriptor address and to receive and transmit data buffer. The extended buffer queue is used to receive the buffer descriptor and update the data buffer, 1588 PTP timestamp buffer address etc. Data receive and transmit is easily managed with this context structure.

Initialization

To initialize the ENET module, follow these steps:

1. First, initialize the ENET MAC and PHY configuration structures, and initialize the upper layer callback function for interrupt mode.
 2. Call the `enet_mac_init()` function with the `enet_mac_api_t` API structure in the `enet_dev_if_t` and pass in the `enet_dev_if_t` device data structure. This function enables the ENET module.
- This is an example code to initialize the ENET device data structure:

```

uint32_t phyAddr;
enet_mac_config_t configMac;
enet_buff_config_t bufferCfg;
enet_config_rmii_t rmiiCfg;
rmiiCfg.duplex = kEnetCfgFullDuplex;
rmiiCfg.speed = kEnetCfgSpeed100M;
rmiiCfg.mode = kEnetCfgRmii;
rmiiCfg.isLoopEnabled = false;
rmiiCfg.isRxOnTxDisabled = false;

configMac.macMode = kEnetMacNormalMode;
configMac.rxMaxFrameLen = kEnetMaxFrameSize;
configMac.rxTruncLen = 2000;
configMac.txInterPacketGap = 9;
configMac.macAddr = source;
configMac.rmiiCfgPtr = &rmiiCfg;
configMac.clkCycle = kEnetMdioHoldTwoClkCycle;
configMac.macCtlConfigure = kEnetRxCrcFwdEnable |
    kEnetTxCrcBdEnable | kEnetMacEnhancedEnable;
configMac.rxFifo.rxAlmostEmpty = kEnetMinFifoAlmostEmpty;
configMac.rxFifo.rxAlmostFull = kEnetMinFifoAlmostEmpty;
configMac.txFifo.txAAlmostEmpty = kEnetMinFifoAlmostEmpty;
configMac.txFifo.txAAlmostFull =
    kEnetDefaultTxFifoAlmostFull;
configMac.txFifo.isStoreForwardEnabled = false;
/* Buffer configure structure*/
memset(&bufferCfg, 0, sizeof(enet_buff_config_t));
bufferCfg.rxBdNumber = ENET_RXBD_NUM;
bufferCfg.rxBdPtrAlign = RxBuffDescrip;
bufferCfg.rxBufferAlign = &RxDataBuff[0][0];
bufferCfg.rxBufferSizeAlign = ENET_RXBufferSizeAlign(ENET_RXBUFF_SIZE);
bufferCfg.txBdNumber = ENET_TXBD_NUM;
bufferCfg.txBdPtrAlign = TxBuffDescrip;
bufferCfg.txBufferAlign = &TxDataBuff[0][0];
bufferCfg.txBuffSizeAlign = ENET_TxBuffSizeAlign(ENET_TXBUFF_SIZE);
#if !ENET_RECEIVE_ALL_INTERRUPT
bufferCfg.extRxBuffQue = &ExtRxDataBuff[0][0];
bufferCfg.extRxBuffNum = ENET_EXTRXBD_NUM;
#endif
#if FSL_FEATURE_ENET_SUPPORT_PTP
configMac.isSlaveMode = false;
bufferCfg.ptpTsRxDataPtr = &ptpTsRxData[0];
bufferCfg.ptpTsRxBuffNum = ENET_PTP_RXTS_RING_LEN;
bufferCfg.ptpTsTxDataPtr = &ptpTsTxData[0];
bufferCfg.ptpTsTxBuffNum = ENET_PTP_TXTS_RING_LEN;
#endif
/* Set up the PHY configuration structure*/
enet_phy_config_t g_enetPhyCfg =
{{0, false }};

/* Initialize ENET callback function for receive interrupt mode*/
#if ENET_RECEIVE_ALL_INTERRUPT
enetIfPtr->enetNetifcall = enet_receive_test;
#endif
/* Initialize ENET device*/
result = ENET_DRV_Init(enetIfPtr, &configMac, &bufferCfg);
/* Initialize PHY*/
if(g_enetPhyCfg[device].isAutodiscoverEnabled)
{

```

ENET Peripheral Driver

```
    uint32_t phyAddr;
    result = PHY_DRV_Autodiscover(device, &phyAddr);
    if(result != kStatus_ENET_Success)
        return result;
    enetIfPtr->phyAddr = phyAddr;
}
else
{
    enetIfPtr->phyAddr = g_enetPhyCfg[device].phyAddr;
}

PHY_DRV_Init(device, enetIfPtr->phyAddr, g_enetPhyCfg[device].isLoopEnabled);
```

Data Receive

ENET can receive data in two different ways. The MACRO ENET_RECEIVE_ALL_INTERRUPT is used to select the way in which data is received.

- interrupt and poll (define ENET_RECEIVE_ALL_INTERRUPT with 0)
- interrupt only (define ENET_RECEIVE_ALL_INTERRUPT with 1)

Interrupt add task poll approach: The ENET driver receives data directly from the buffer descriptor to the upper layer. To shorten the receive process time on the receive interrupt, the receive data uses the interrupt and poll combination:

1. Receive interrupt: releases the receive synchronize signal (enetReceiveSync).
2. Poll: receives task and waits for the receive synchronize signal when no data is received. The receive data returns the address and data length of the received data. To receive data from the ENET device, create a receive task as follows:

```
/* Create the task when do net device initialize*/
task_create(ENET_receive, ENET_TEST_TASK_PRIO, enetIfPtr, &revHandle);

.....



ENET_receive(void *param)
{
    uint8_t *packet;
    uint16_t length;
    enet_mac_packet_buffer_t packetBuffer[
        kENetMaxFrameBdNumbers];

    enet_dev_if_t * enetIfPtr = (enet_dev_if_t *)param;
    while(1)
    {
        /* Receive frame*/
        result = ENET_DRV_ReceiveData(enetIfPtr, packetBuffer);
        if ((result == kStatus_ENET_RxbdEmpty) || (result ==
            kStatus_ENET_InvalidInput))
#if !USERTOS
        {
            status = OSA_EventWait(&enetIfPtr->enetReceiveSync, flag, false, 0, &flagCheck);
        }
        else if(result == kStatus_ENET_Success)
        {
#endif
        {
            OSA_EventWait(&enetIfPtr->enetReceiveSync, flag, false,
OSA_WAIT_FOREVER, &flagCheck);
```

```

        }
#endif

.....
/* The packets delivery to upper layer*/
.....
}

}

/* Receive interrupt handler to wake up blocked receive task*/
void ENET_DRV_RxIRQHandler(uint32_t instance)
{
    enet_dev_if_t *enetIfPtr;
    uint32_t baseAddr;
    enetIfPtr = enetIfHandle[instance];
    event_flags_t flag = 0x1;
    /*Check input parameter*/
    if (!enetIfPtr)
    {
        return;
    }
    baseAddr = g_enetBaseAddr[enetIfPtr->deviceNumber];
    /* Get interrupt status.*/
    while ((ENET_HAL_GetIntStatusFlag(baseAddr,
        kEnetRxFrameInterrupt)) || (ENET_HAL_GetIntStatusFlag(
        baseAddr, kEnetRxByteInterrupt)))
    {
        /*Clear interrupt*/
        ENET_HAL_ClearIntStatusFlag(baseAddr,
        kEnetRxFrameInterrupt);
        ENET_HAL_ClearIntStatusFlag(baseAddr,
        kEnetRxByteInterrupt);
        /* Release sync signal-----*/
        OSA_EventSet(&enetIfPtr->enetReceiveSync, flag);
    }
}

Interrupt only approach for Mac receive:
The ENET driver receives data directly from the buffer descriptor to the upper layer.
In this case, data is received on the receive interrupt handler:
1. Receive interrupt handler calls the receive peripheral driver.
2. Receive peripheral driver calls the initialized callback function.
3. The callback function checks the protocol and delivers the received data to the upper layer of the
TCP/IP stack.

These are the details:
~~~~~{.c}

void ENET_DRV_RxIRQHandler(uint32_t instance)
{
    enet_dev_if_t *enetIfPtr;
    uint32_t baseAddr;
    enetIfPtr = enetIfHandle[instance];

    /*Check input parameter*/
    if (!enetIfPtr)
    {
        return;
    }
    baseAddr = g_enetBaseAddr[enetIfPtr->deviceNumber];
    /* Get interrupt status.*/
    while ((ENET_HAL_GetIntStatusFlag(baseAddr,
        kEnetRxFrameInterrupt)) || (ENET_HAL_GetIntStatusFlag(
        baseAddr, kEnetRxByteInterrupt)))

```

ENET Peripheral Driver

```
{  
    /*Clear interrupt*/  
    ENET_HAL_ClearIntStatusFlag(baseAddr,  
        kEnetRxFrameInterrupt);  
    ENET_HAL_ClearIntStatusFlag(baseAddr,  
        kEnetRxByteInterrupt);  
    /* Receive peripheral driver*/  
    ENET_DRV_ReceiveData(enetIfPtr);  
}  
  
}  
  
.....  
  
uint32_t ENET_DRV_ReceiveData(enet_dev_if_t * enetIfPtr)  
{  
    void *curBd;  
    uint32_t length;  
    uint8_t *packet;  
    uint32_t controlStatus;  
  
    /* Check input parameters*/  
    if(!enetIfPtr)  
    {  
        return kStatus_ENET_InvalidInput;  
    }  
  
    .....  
  
    /* callback function to delivery to stack upper layer */  
    enetIfPtr->enetNetifcall(enetIfPtr, packet, length);  
  
    .....  
  
    return kStatus_ENET_Success;  
}  
  
uint32_t void enet_callback(void *param, enet_mac_packet_buffer_t *packetBuffer)  
{  
    uint32_t length = 0;  
    uint16_t type, counter = 0;  
    uint8_t *packet;  
  
    /* Collect the frame first*/  
    if(!packetBuffer[1].length)  
    {  
        packet = packetBuffer[0].data; /* the frame with only one bd */  
        length = packetBuffer[0].length;  
    }  
    else  
    {  
        /* Dequeue a large buffer */  
        packet = enet_mac_dequeue_buffer((void **) &dataBuffQue);  
        if(packet!=NULL)  
        {  
            for(counter = 0; packetBuffer[counter].next != NULL ; counter ++)  
            {  
                memcpy(packet + length, packetBuffer[counter].data, packetBuffer[counter].length);  
                length += packetBuffer[counter].length;  
            }  
        }  
        else  
        {  
    }
```

```

        return kStatus_ENET_LargeBufferFull;
    }
}
/* Process the received frame*/
type = ntohs(*(uint16_t *)&((enet_etherent_header_t *)packet)->type);
/* Collect frame to PCB structure for upper layer process*/
QUEUEGET(packbuffer[enetIfPtr->deviceNumber].pcbHead, packbuffer[enetIfPtr->
    deviceNumber].pcbTail, pcbPtr);
if(pcbPtr)
{
    pcbPtr->FRAG[0].LENGTH = length;
    pcbPtr->FRAG[0].FRAGMENT = packet;
    pcbPtr->PRIVATE = (void *)enetIfPtr;

    switch(type)
    {
        case ENETPROT_IP:
            IPE_recv_IP((PCB *)pcbPtr, enetIfPtr->netIfPtr);
            break;
        case ENETPROT_ARP:
            IPE_recv_ARP((PCB *)pcbPtr, enetIfPtr->netIfPtr);
            break;
        case ENETPROT_IP6:
            IP6E_recv_IP((PCB *)pcbPtr, enetIfPtr->netIfPtr);
            break;
        case ENETPROT_ETHERNET:
            enet_ptp_service_l2packet(enetIfPtr, packet, length);
            break;
        default:
            PCB_free((PCB *)pcbPtr);
            break;
    }
}
else
{
    enetIfPtr->stats.statsRxMissed++;
}
}

/* Remember to enqueue the packet to the dataBuffQue and enqueue the pcbPtr to the packbuffer
   after the data is processed by the upper layer. This process is done in the ENET_free API
   and called by the RTCS. */

```

Data Transmit

Data Transmit transmits data from the TCP/IP layer to the device and, then, to the network. The data transmit function should be used like this: PCB_PTR is the data buffer structure of the frame which contains many PCB_FRAGMENT(including the data buffer and length).

```

uint32_t ENET_send(_enet_handle handle, PCB_PTR packet, uint32_t type,
    _enet_address dest, uint32_t flags)
{
    uint8_t headerLen, *frame;
    PCB_FRAGMENT *fragPtr;
    uint16_t size = 0, lenTemp = 0, bdNumUsed = 0;
    enet_dev_if_t *enetIfPtr;
    enet_etherent_header_t *packetPtr;
    volatile enet_bd_struct_t * curBd;
    uint32_t result, lenoffset = 0;
    /*Check out*/
    if ((!handle) || (!packet))

```

ENET Peripheral Driver

```
{  
    return kStatus_ENET_InvalidInput;  
}  
  
enetIfPtr = (enet_dev_if_t *)handle;  
/* Default frame header size*/  
headerLen = kENetEthernetHeadLen;  
  
/* Check the frame length*/  
for (fragPtr = packet->FRAG; fragPtr->LENGTH; fragPtr++)  
{  
    size += fragPtr->LENGTH;  
}  
if (size > enetIfPtr->maxFrameSize)  
{  
    return kStatus_ENET_TxLarge;  
}  
  
/*Add MAC hardware address*/  
packetPtr = (enet_ethernet_header_t *)packet->FRAG[0].FRAGMENT;  
htone(packetPtr->destAddr, dest);  
htone(packetPtr->sourceAddr, enetIfPtr->macAddr);  
packetPtr->type = HTONS(type);  
if (flags & ENET_OPT_8021QTAG)  
{  
    enet_8021vlan_header_t *vlanHeadPtr = (  
        enet_8021vlan_header_t *)packetPtr;  
    vlanHeadPtr->tpidtag = HTONS(ENETPROT_8021Q);  
    vlanHeadPtr->othertag = HTONS((ENET_GETOPT_8021QPRIO(flags) << 13));  
    vlanHeadPtr->type = HTONS(type);  
    headerLen = sizeof(enet_8021vlan_header_t);  
    packet->FRAG[0].LENGTH = headerLen;  
}  
  
if (flags & ENET_OPT_8023)  
{  
    enet_8022_header_ptr lcPtr = (enet_8022_header_ptr)(packetPtr->type + 2);  
    packetPtr->type = HTONS(size - headerLen);  
    lcPtr->dsap[0] = 0xAA;  
    lcPtr->ssap[0] = 0xAA;  
    lcPtr->command[0] = 0x03;  
    lcPtr->oui[0] = 0x00;  
    lcPtr->oui[1] = 0x00;  
    lcPtr->oui[2] = 0x00;  
    lcPtr->type = HTONS(type);  
    packet->FRAG[0].LENGTH = packet->FRAG[0].LENGTH+ sizeof(  
        enet_8022_header_t);  
}  
  
/* Get the current transmit data buffer in buffer descriptor */  
curBd = enetIfPtr->bdContext.txBdCurPtr;  
frame = ENET_HAL_GetBuffDescripData(curBd);  
  
/* Send a whole frame with a signal buffer*/  
if(size <= enetIfPtr->bdContext.txBuffSizeAlign)  
{  
    bdNumUsed = 1;  
    for (fragPtr = packet->FRAG; fragPtr->LENGTH; fragPtr++)  
    {  
        memcpy(frame + lenTemp, fragPtr->FRAGMENT, fragPtr->LENGTH);  
        lenTemp += fragPtr->LENGTH;  
    }  
    /* Send packet to the device*/  
    return ENET_DRV_SendData(enetIfPtr, size, bdNumUsed);  
}  
  
/* Copy the Ethernet header first*/  
memcpy(frame, packet->FRAG[0].FRAGMENT, packet->FRAG[0].LENGTH);
```

```

/* Send a whole frame with multiple buffer descriptors*/
while((size - bdNumUsed* enetIfPtr->bdContext.txBuffSizeAlign) > enetIfPtr->
      bdContext.txBuffSizeAlign)
{
    if(bdNumUsed == 0)
    {
        memcpy((void *) (frame + packet->FRAG[0].LENGTH), (void *) (packet->FRAG[1].FRAGMENT), enetIfPtr
->bdContext.txBuffSizeAlign - packet->FRAG[0].LENGTH);
        lenoffset += (enetIfPtr->bdContext.txBuffSizeAlign - packet->FRAG[0].
LENGTH);
    }
    else
    {
        memcpy((void *)frame, (void *) (packet->FRAG[1].FRAGMENT + lenoffset), enetIfPtr->
bdContext.txBuffSizeAlign);
        lenoffset += enetIfPtr->bdContext.txBuffSizeAlign;
    }

    /* Incremenet the buffer descriptor*/
    curBd = ENET_DRV_IncrTxBuffDescripIndex(enetIfPtr, curBd);
    frame = ENET_HAL_GetBuffDescripData(curBd);
    /* Increment the index and parameters*/
    bdNumUsed++;
}

memcpy((void *)frame, (void *) (packet->FRAG[1].FRAGMENT + lenoffset), size - bdNumUsed * enetIfPtr->
      bdContext.txBuffSizeAlign);
bdNumUsed++;
/* Send packet to the device*/
result = ENET_DRV_SendData(enetIfPtr, size, bdNumUsed);

/* Free the PCB buffer if nessary*/
PCB_free(packet);

return result;
}

```

#Note:

- TWR-MK70F120M board only routes the RMII interface signals from the CPU board to the primary connectors. Hence, the rmiiCfgMode in the `enet_mac_config_t` should be set to the kEnetCfgRmii. TWR-MK64F120M board can configure both RMII and MII modes.
- Jumper setting When ENET is uses the RMII interface signals by default, the RMII input clock must be kept in phase with the clock supplied to the external PHY. Jumpers should be set like this:
 - TWR-SER J2 shunt across 3-4 , J3 shunt across 2-3, J12 shunt across 9-10
 - TWR-MK64f120M board make sure J32 jumper is on to disable the OSC on the CPU board.
 - TWR-MK70f120M board make sure J18 jumper is on to disable the OSC on the CPU board.
When ENET chooses the MII interface signals, the jumper should be set as: TWR-SER J2 shunt across 1-2, J12 no shunt across 9-10.

10.2.1 Data Structure Documentation

10.2.1.1 struct enet_multicast_group_t

Data Fields

- uint8_t `groupAddr [kEnetMacAddrLen]`

ENET Peripheral Driver

- *Multicast group address.*
• `uint32_t hash`
Hash value of the multicast group address.
- `struct ENETMulticastGroup * next`
Pointer of the next group structure.
- `struct ENETMulticastGroup * prv`
Pointer of the previous structure.

10.2.1.2 struct enet_ethernet_header_t

Data Fields

- `uint8_t destAddr [kEnetMacAddrLen]`
Destination address.
- `uint8_t sourceAddr [kEnetMacAddrLen]`
Source address.
- `uint16_t type`
Protocol type.

10.2.1.3 struct enet_8021vlan_header_t

Data Fields

- `uint8_t destAddr [kEnetMacAddrLen]`
Destination address.
- `uint8_t sourceAddr [kEnetMacAddrLen]`
Source address.
- `uint16_t tpidtag`
ENET 8021tag header tag region.
- `uint16_t othertag`
ENET 8021tag header type region.
- `uint16_t type`
Protocol type.

10.2.1.4 struct enet_buff_descrip_context_t

Data Fields

- `volatile enet_bd_struct_t * rxBdBasePtr`
Receive buffer descriptor base address pointer.
- `volatile enet_bd_struct_t * rxBdCurPtr`
Current receive buffer descriptor pointer.
- `volatile enet_bd_struct_t * rxBdDirtyPtr`
Receive dirty buffer descriptor.
- `volatile enet_bd_struct_t * txBdBasePtr`
Transmit buffer descriptor base address pointer.
- `volatile enet_bd_struct_t * txBdCurPtr`
Current transmit buffer descriptor pointer.

- volatile `enet_bd_struct_t * txBdDirtyPtr`
Last cleaned transmit buffer descriptor pointer.
- bool `isTxBdFull`
Transmit buffer descriptor full.
- bool `isRxBdFull`
Receive buffer descriptor full.
- uint32_t `rxBuffSizeAlign`
Receive buffer size alignment.
- uint32_t `txBuffSizeAlign`
Transmit buffer size alignment.
- bool `isTxCrcEnable`
Transmit CRC enable in buffer descriptor.
- bool `isRxCrcFwdEnable`
Receive CRC forward.
- uint8_t * `extRxBuffQue`
Extended Rx data buffer queue to update the data buff in the receive buffer descriptor.
- uint8_t `extRxBuffNum`
extended data buffer number

10.2.1.5 struct enet_stats_t

Data Fields

- uint32_t `statsRxTotal`
Total number of receive packets.
- uint32_t `statsTxTotal`
Total number of transmit packets.

10.2.1.6 struct enet_mac_packet_buffer_t

Data Fields

- uint8_t * `data`
Data buffer pointer.
- uint16_t `length`
Data length.
- struct ENETMacPacketBuffer * `next`
Next pointer.

10.2.1.7 struct enet_buff_config_t

Data Fields

- uint16_t `rxBdNumber`
Receive buffer descriptor number.
- uint16_t `txBdNumber`
Transmit buffer descriptor number.
- uint32_t `rxBuffSizeAlign`

ENET Peripheral Driver

- `aligned receive buffer size and must be larger than 256.`
• `uint32_t txBuffSizeAlign`
- `aligned transmit buffer size and must be larger than 256.`
• `volatile enet_bd_struct_t * rxBdPtrAlign`
Aligned receive buffer descriptor pointer.
- `uint8_t * rxBufferAlign`
Aligned receive data buffer pointer.
- `volatile enet_bd_struct_t * txBdPtrAlign`
Aligned transmit buffer descriptor pointer.
- `uint8_t * txBufferAlign`
Aligned transmit buffer descriptor pointer.
- `uint8_t * extRxBuffQue`
Extended Rx data buffer queue to update the data buff in the receive buffer descriptor.
- `uint8_t extRxBuffNum`
extended data buffer number

10.2.1.8 struct enet_dev_if_t

Data Fields

- `struct ENETDevIf * next`
Next device structure address.
- `void * netIfPrivate`
For upper layer private structure.
- `enet_multicast_group_t * multiGroupPtr`
Multicast group chain.
- `uint32_t deviceNumber`
Device number.
- `uint8_t macAddr [kEnetMacAddrLen]`
Mac address.
- `uint8_t phyAddr`
PHY address connected to this device.
- `bool isInitialized`
Device initialized.
- `uint16_t maxFrameSize`
Mac maximum frame size.
- `bool isVlanTagEnabled`
ENET VLAN-TAG frames enabled.
- `enet_buff_descrip_context_t bdContext`
Mac buffer descriptors context pointer.
- `enet_stats_t stats`
Packets statistic.
- `enet_netif_callback_t enetNetifcall`
Receive callback function to the upper layer.
- `mutex_t enetContextSync`
Sync signal.

10.2.2 Macro Definition Documentation

10.2.2.1 #define ENET_ENABLE_DETAIL_STATS 0

10.2.2.2 #define ENET_ORIGINAL_CRC32 0xFFFFFFFFU

CRC-32 Original data

10.2.3 Enumeration Type Documentation

10.2.3.1 enum enet_crc_parameter_t

Enumerator

kEnetCrcOffset CRC-32 offset2.

kEnetCrcMask1 CRC-32 mask.

10.2.3.2 enum enet_protocol_type_t

Enumerator

kEnetProtocol2ptpv2 Packet type Ethernet ieee802.3.

kEnetProtocolIpv4 Packet type IPv4.

kEnetProtocolIpv6 Packet type IPv6.

kEnetProtocol8021QVlan Packet type VLAN.

kEnetPacketUdpVersion UDP protocol type.

kEnetPacketIpv4Version Packet IP version IPv4.

kEnetPacketIpv6Version Packet IP version IPv6.

10.2.4 Function Documentation

10.2.4.1 uint32_t ENET_DRV_Init (enet_dev_if_t * enetIfPtr, const enet_mac_config_t * macCfgPtr, enet_buff_config_t * buffCfgPtr)

Parameters

<i>enetIfPtr</i>	The pointer to the basic Ethernet structure.
------------------	--

ENET Peripheral Driver

<i>macCfgPtr</i>	The Mac configure structure pointer.
<i>buffCfgPtr</i>	The buffer configure structure pointer.

Returns

The execution status.

10.2.4.2 uint32_t ENET_DRV_Deinit (*enet_dev_if_t * enetIfPtr*)

Parameters

<i>enetIfPtr</i>	The ENET context structure.
------------------	-----------------------------

Returns

The execution status.

10.2.4.3 uint32_t ENET_DRV_UpdateRxBuffDescrip (*enet_dev_if_t * enetIfPtr, bool isBuffUpdate*)

This function updates the used receive buffer descriptor ring to ensure that the used BDS is correctly used again. It cleans the status region and sets the control region of the used receive buffer descriptor. If the isBufferUpdate flag is set, the data buffer in the buffer descriptor is updated.

Parameters

<i>enetIfPtr</i>	The ENET context structure.
<i>isBufferUpdate</i>	The data buffer update flag.

Returns

The execution status.

10.2.4.4 uint32_t ENET_DRV_CleanupTxBuffDescrip (*enet_dev_if_t * enetIfPtr*)

First, store the transmit frame error statistic and PTP timestamp of the transmitted packets. Second, clean up the used transmit buffer descriptors. If the PTP 1588 feature is open, this interface captures the 1588 timestamp. It is called by the transmit interrupt handler.

Parameters

<i>enetIfPtr</i>	The ENET context structure.
------------------	-----------------------------

Returns

The execution status.

10.2.4.5 **volatile enet_bd_struct_t* ENET_DRV_IncrRxBuffDescripIndex (enet_dev_if_t * *enetIfPtr*, volatile enet_bd_struct_t * *curBd*)**

Parameters

<i>enetIfPtr</i>	The ENET context structure.
<i>curBd</i>	The current buffer descriptor pointer.

Returns

the increased received buffer descriptor.

10.2.4.6 **volatile enet_bd_struct_t* ENET_DRV_IncrTxBuffDescripIndex (enet_dev_if_t * *enetIfPtr*, volatile enet_bd_struct_t * *curBd*)**

Parameters

<i>enetIfPtr</i>	The ENET context structure.
<i>curBd</i>	The current buffer descriptor pointer.

Returns

the increased transmit buffer descriptor.

10.2.4.7 **bool ENET_DRV_RxErrorStats (enet_dev_if_t * *enetIfPtr*, uint32_t *data*)**

This interface gets the error statistics of the received frame. Because the error information is in the last BD of a frame, this interface should be called when processing the last BD of a frame.

ENET Peripheral Driver

Parameters

<i>enetIfPtr</i>	The ENET context structure.
<i>data</i>	The current control and status data of the buffer descriptor.

Returns

The frame error status.

- True if the frame has an error.
- False if the frame does not have an error.

10.2.4.8 void ENET_DRV_TxErrorStats (*enet_dev_if_t * enetIfPtr, volatile enet_bd_struct_t * curBd*)

This interface gets the error statistics of the transmit frame. Because the error information is in the last BD of a frame, this interface should be called when processing the last BD of a frame.

Parameters

<i>enetIfPtr</i>	The ENET context structure.
<i>curBd</i>	The current buffer descriptor.

10.2.4.9 uint32_t ENET_DRV_ReceiveData (*enet_dev_if_t * enetIfPtr*)

Parameters

<i>enetIfPtr</i>	The ENET context structure.
------------------	-----------------------------

Returns

The execution status.

10.2.4.10 uint32_t ENET_DRV_SendData (*enet_dev_if_t * enetIfPtr, uint32_t dataLen, uint32_t bdNumUsed*)

Parameters

<i>enetIfPtr</i>	The ENET context structure.
<i>dataLen</i>	The frame data length to be transmitted.
<i>bdNumUsed</i>	The buffer descriptor need to be used.

Returns

The execution status.

10.2.4.11 void ENET_DRV_RxIRQHandler (uint32_t *instance*)

Parameters

<i>instance</i>	The ENET instance number.
-----------------	---------------------------

10.2.4.12 void ENET_DRV_TxIRQHandler (uint32_t *instance*)

Parameters

<i>instance</i>	The ENET instance number.
-----------------	---------------------------

10.2.4.13 void ENET_DRV_CalculateCrc32 (uint8_t * *address*, uint32_t * *crcValue*)

Parameters

<i>address</i>	The ENET Mac hardware address.
<i>crcValue</i>	The calculated CRC value of the Mac address.

10.2.4.14 uint32_t ENET_DRV_AddMulticastGroup (uint32_t *instance*, uint8_t * *address*, uint32_t * *hash*)

Parameters

<i>instance</i>	The ENET instance number.
<i>multiGroupPtr</i>	The ENET multicast group structure.
<i>address</i>	The ENET Mac hardware address.

ENET Peripheral Driver

Returns

The execution status.

**10.2.4.15 uint32_t ENET_DRV_LeaveMulticastGroup (uint32_t *instance*, uint8_t *
address)**

Parameters

<i>instance</i>	The ENET instance number.
<i>multiGroupPtr</i>	The ENET multicast group structure.
<i>address</i>	The ENET Mac hardware address.

Returns

The execution status.

10.2.4.16 void enet_mac_enqueue_buffer (void *queue*, void **buffer*)**

Parameters

<i>queue</i>	The buffer queue address.
<i>buffer</i>	The buffer will add to the buffer queue.

10.2.4.17 void* enet_mac_dequeue_buffer (void *queue*)**

Parameters

<i>queue</i>	The buffer queue address.
<i>buffer</i>	The buffer will remove from the buffer queue.

10.3 ENET RTCS Adaptor

This chapter describes the programming interface of the ENET RTCS Adaptor.

Data Structures

- struct [ENET_HEADER_PTR](#)
Definition of the Ethernet packet header structure. [More...](#)
- struct [PCB_FRAGMENT_PTR](#)
Definition of the fragment PCB structure. [More...](#)
- struct [PCB_PTR](#)
Definition of the PCB structure for the RTCS adaptor. [More...](#)
- struct [PCB2_PTR](#)
Definition of the two fragment PCB structure. [More...](#)
- struct [pcb_queue](#)
Definition of the two fragment PCB structure. [More...](#)
- struct [enet_ecb_struct_t](#)
Definition of the ECB structure, which contains the protocol type and it's related service function. [More...](#)
- struct [enet_8022_header_ptr](#)
Definition of the 8022 header. [More...](#)
- struct [ENET_COMMON_STATS_STRUCT_PTR](#)
Definition of the common status structure. [More...](#)

Macros

- #define [ENET_RXBD_NUM](#) (8)
Definitions of the configuration parameter.
- #define [ENET_OK](#) (0)
Definitions of the error codes.
- #define [ENETPROT_IP](#) 0x0800
Definitions of the ENET protocol parameter.
- #define [ENET_OPTION_HW_TX_IP_CHECKSUM](#) 0x00001000
Definitions of the ENET option macro.
- #define [ENET_DEFAULT_MAC_ADD](#) { 0x00, 0x00, 0x5E, 0, 0, 0 }
Definitions of the ENET default Mac.
- #define [RTCS_HTONS](#)(n) (n)
Definitions of the macro for byte-swap.
- #define [QUEUEADD](#)(head, tail, pcb)
Definitions of the add to queue.
- #define [QUEUEGET](#)(head, tail, pcb)
Definitions of the get from queue.

Typedefs

- typedef unsigned char [_enet_address](#) [6]
Definition for ENET six-byte Mac type.
- typedef void * [_enet_handle](#)
Definition of the IPCFG structure.

ENET RTCS Adaptor

Variables

- unsigned long `_RTCSTASK_priority`
Definitions of the task parameter.

ENET RTCS ADAPTOR

- `uint32_t ENET_initialize (uint32_t device, _enet_address address, uint32_t flag, _enet_handle *handle)`
Initializes the ENET device.
- `uint32_t ENET_open (_enet_handle handle, uint16_t type, void(*service)(PCB_PTR, void *), void *private)`
Opens the ENET device.
- `uint32_t ENET_shutdown (_enet_handle handle)`
Shuts down the ENET device.
- `static void ENET_receive (task_param_t param)`
ENET frame receive.
- `uint32_t ENET_send (_enet_handle handle, PCB_PTR packet, uint32_t type, _enet_address dest, uint32_t flags)`
ENET frame transmit.
- `uint32_t ENET_get_address (_enet_handle handle, _enet_address address)`
The ENET gets the address with the initialized device.
- `uint32_t ENET_get_mac_address (uint32_t device, uint32_t value, _enet_address address)`
The ENET gets the address with an uninitialized device.
- `uint32_t ENET_join (_enet_handle handle, uint16_t type, _enet_address address)`
The ENET joins a multicast group address.
- `uint32_t ENET_leave (_enet_handle handle, uint16_t type, _enet_address address)`
The ENET leaves a multicast group address.
- `bool ENET_link_status (_enet_handle handle)`
The ENET gets the link status.
- `uint32_t ENET_get_speed (_enet_handle handle)`
The ENET gets the link speed.
- `uint32_t ENET_get_MTU (_enet_handle handle)`
The ENET gets the MTU.
- `bool ENET_phy_registers (_enet_handle handle, uint32_t numRegs, uint32_t *regPtr)`
Gets the ENET PHY registers.
- `uint32_t ENET_get_options (_enet_handle handle)`
Gets ENET options.
- `uint32_t ENET_close (_enet_handle handle, uint16_t type)`
Registers a protocol type on an Ethernet channel.
- `uint32_t ENET_mediactl (_enet_handle handle, uint32_t commandId, void *inOutParam)`
ENET mediactl.
- `_enet_handle ENET_get_next_device_handle (_enet_handle handle)`
Gets the next ENET device handle address.
- `void ENET_free (PCB_PTR packet)`
ENET free.
- `const char * ENET_strerror (uint32_t error)`
ENET error description.

10.3.1 Data Structure Documentation

10.3.1.1 struct ENET_HEADER

Data Fields

- `_enet_address DEST`
destination Mac address
- `_enet_address SOURCE`
source Mac address
- `unsigned char TYPE [2]`
protocol type

10.3.1.2 struct PCB_FRAGMENT

Data Fields

- `uint32_t LENGTH`
Packet fragment length.
- `unsigned char * FRAGMENT`
brief Pointer to fragment

10.3.1.3 struct PCB

Data Fields

- `PCB_FREE_FPTR FREE`
Function that frees PCB.
- `void * PRIVATE`
Private PCB information.
- `PCB_FRAGMENT FRAG [1]`
Pointer to PCB fragment.

10.3.1.4 struct PCB2

Data Fields

- `PCB_FREE_FPTR FREE`
Function that frees PCB.
- `void * PRIVATE`
Private PCB information.
- `PCB_FRAGMENT FRAG [2]`
Pointers to two PCB fragments.

10.3.1.5 struct pcb_queue

Data Fields

- PCB * `pcbHead`
PCB buffer head.
- PCB * `pcbTail`
PCB buffer tail.

10.3.1.6 struct enet_ecb_struct_t

10.3.1.7 struct enet_8022_header_t

Data Fields

- uint8_t `dsap` [1]
DSAP region.
- uint8_t `ssap` [1]
SSAP region.
- uint8_t `command` [1]
Command region.
- uint8_t `oui` [3]
OUI region.
- uint16_t `type`
type region

10.3.1.8 struct ENET_COMMON_STATS_STRUCT

Data Fields

- uint32_t `ST_RX_TOTAL`
Total number of received packets.
- uint32_t `ST_RX_MISSED`
Number of missed packets.
- uint32_t `ST_RX_DISCARDED`
Discarded a protocol that was not recognized.
- uint32_t `ST_RX_ERRORS`
Discarded error during reception.
- uint32_t `ST_TX_TOTAL`
Total number of transmitted packets.
- uint32_t `ST_TX_MISSED`
Discarded transmit ring full.
- uint32_t `ST_TX_DISCARDED`
Discarded bad packet.
- uint32_t `ST_TX_ERRORS`
Error during transmission.

10.3.2 Function Documentation

10.3.2.1 `uint32_t ENET_initialize(uint32_t device, _enet_address address, uint32_t flag, _enet_handle * handle)`

ENET RTCS Adaptor

Parameters

<i>device</i>	The ENET device number.
<i>address</i>	The hardware address.
<i>flag</i>	The flag for upper layer.
<i>handle</i>	The address pointer for ENET device structure.

Returns

The execution status.

10.3.2.2 `uint32_t ENET_open (_enet_handle handle, uint16_t type, void(*)(PCB_PTR, void *) service, void * private)`

Parameters

<i>handle</i>	The address pointer for ENET device structure.
<i>type</i>	The ENET protocol type.
<i>service</i>	The service function for type.
<i>private</i>	The private data for ENET device.

Returns

The execution status.

10.3.2.3 `uint32_t ENET_shutdown (_enet_handle handle)`

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

The execution status.

10.3.2.4 `static void ENET_receive (task_param_t param) [static]`

Parameters

<i>enetIfPtr</i>	The address pointer for ENET device structure.
------------------	--

10.3.2.5 **uint32_t ENET_send (_enet_handle *handle*, PCB_PTR *packet*, uint32_t *type*, _enet_address *dest*, uint32_t *flags*)**

Parameters

<i>handle</i>	The address pointer for ENET device structure.
<i>packet</i>	The ENET packet buffer.
<i>type</i>	The ENET protocol type.
<i>dest</i>	The destination hardware address.
<i>flag</i>	The flag for upper layer.

Returns

The execution status.

10.3.2.6 **uint32_t ENET_get_address (_enet_handle *handle*, _enet_address *address*)**

Parameters

<i>handle</i>	The address pointer for ENET device structure.
<i>address</i>	The destination hardware address.

Returns

The execution status.

10.3.2.7 **uint32_t ENET_get_mac_address (uint32_t *device*, uint32_t *value*, _enet_address *address*)**

ENET RTCS Adaptor

Parameters

<i>handle</i>	The address pointer for ENET device structure.
<i>value</i>	The value to change the last three bytes of hardware.
<i>address</i>	The destination hardware address.

Returns

True if the execution status is success else false.

10.3.2.8 **uint32_t ENET_join (_enet_handle *handle*, uint16_t *type*, _enet_address *address*)**

Parameters

<i>handle</i>	The address pointer for ENET device structure.
<i>type</i>	The ENET protocol type.
<i>address</i>	The destination hardware address.

Returns

The execution status.

10.3.2.9 **uint32_t ENET_leave (_enet_handle *handle*, uint16_t *type*, _enet_address *address*)**

Parameters

<i>handle</i>	The address pointer for ENET device structure.
<i>type</i>	The ENET protocol type.
<i>address</i>	The destination hardware address.

Returns

The execution status.

10.3.2.10 **bool ENET_link_status (_enet_handle *handle*)**

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

The link status.

10.3.2.11 uint32_t ENET_get_speed (_enet_handle handle)

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

The link speed.

10.3.2.12 uint32_t ENET_get_MTU (_enet_handle handle)

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

The link MTU

10.3.2.13 bool ENET_phy_registers (_enet_handle handle, uint32_t numRegs, uint32_t * regPtr)

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

ENET RTCS Adaptor

<i>numRegs</i>	The number of registers.
<i>regPtr</i>	The buffer for data read from PHY registers.

Returns

True if all numRegs registers are read succeed else false.

10.3.2.14 uint32_t ENET_get_options (_enet_handle *handle*)

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

ENET options.

10.3.2.15 uint32_t ENET_close (_enet_handle *handle*, uint16_t *type*)

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

ENET options.

10.3.2.16 uint32_t ENET_mediactl (_enet_handle *handle*, uint32_t *commandId*, void * *inOutParam*)

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

<i>The</i>	command ID.
<i>The</i>	buffer for input or output parameters.

Returns

ENET options.

10.3.2.17 `_enet_handle ENET_get_next_device_handle (_enet_handle handle)`

Parameters

<i>handle</i>	The address pointer for ENET device structure.
---------------	--

Returns

The address of next ENET device handle.

10.3.2.18 `void ENET_free (PCB_PTR packet)`

Parameters

<i>packet</i>	The buffer address.
---------------	---------------------

10.3.2.19 `const char* ENET_strerror (uint32_t error)`

Parameters

<i>error</i>	The ENET error code.
--------------	----------------------

Returns

The error string.

10.4 ENET Physical Layer Driver

This chapter describes the programming interface of the ENET Physical Layer Driver.

Chapter 11

FlexTimer (FTM)

The Kinetis SDK provides both HAL and Peripheral drivers for the FlexTimer (FTM) block of Kinetis devices.

Modules

- [FlexTimer HAL driver](#)

This part describes the programming interface of the FlexTimer HAL driver.

- [FlexTimer Peripheral Driver](#)

This part describes the programming interface of the FlexTimer Peripheral driver.

FlexTimer HAL driver

11.1 FlexTimer HAL driver

This chapter describes the programming interface of the FlexTimer HAL driver.

Data Structures

- union `ftm_edge_mode_t`
FlexTimer edge mode. [More...](#)
- struct `ftm_pwm_param_t`
FlexTimer driver PWM parameter. [More...](#)
- struct `ftm_phase_params_t`
FlexTimer quadrature decode phase parameters. [More...](#)

Macros

- `#define HW_CHAN0 (0U)`
Channel number for CHAN0.
- `#define HW_CHAN1 (1U)`
Channel number for CHAN1.
- `#define HW_CHAN2 (2U)`
Channel number for CHAN2.
- `#define HW_CHAN3 (3U)`
Channel number for CHAN3.
- `#define HW_CHAN4 (4U)`
Channel number for CHAN4.
- `#define HW_CHAN5 (5U)`
Channel number for CHAN5.
- `#define HW_CHAN6 (6U)`
Channel number for CHAN6.
- `#define HW_CHAN7 (7U)`
Channel number for CHAN7.

Enumerations

- enum `ftm_clock_source_t`
FlexTimer clock source selection.
- enum `ftm_counting_mode_t`
FlexTimer counting mode selection.
- enum `ftm_clock_ps_t`
FlexTimer pre-scaler factor selection for the clock source.
- enum `ftm_deadtime_ps_t`
FlexTimer pre-scaler factor for the deadtime insertion.
- enum `ftm_config_mode_t`
FlexTimer operation mode, capture, output, dual.
- enum `ftm_input_capture_edge_mode_t`
FlexTimer input capture edge mode, rising edge, or falling edge.
- enum `ftm_output_compare_edge_mode_t`
FlexTimer output compare edge mode.

- enum `ftm_pwm_edge_mode_t`
FlexTimer PWM output pulse mode, high-true or low-true on match up.
- enum `ftm_dual_capture_edge_mode_t`
FlexTimer dual capture edge mode, one shot or continuous.
- enum `ftm_quad_decode_mode_t`
FlexTimer quadrature decode modes, phase encode or count and direction mode.
- enum `ftm_quad_phase_polarity_t`
FlexTimer quadrature phase polarities, normal or inverted polarity.

Functions

- static void `FTM_HAL_SetClockSource` (uint32_t `ftmBaseAddr`, `ftm_clock_source_t` `clock`)
Sets the FTM clock source.
- static uint8_t `FTM_HAL_GetClockSource` (uint32_t `ftmBaseAddr`)
Reads the FTM clock source.
- static void `FTM_HAL_SetClockPs` (uint32_t `ftmBaseAddr`, `ftm_clock_ps_t` `ps`)
Sets the FTM clock divider.
- static uint8_t `FTM_HAL_GetClockPs` (uint32_t `ftmBaseAddr`)
Reads the FTM clock divider.
- static void `FTM_HAL_EnableTimerOverflowInt` (uint32_t `ftmBaseAddr`)
Enables the FTM peripheral timer overflow interrupt.
- static void `FTM_HAL_DisableTimerOverflowInt` (uint32_t `ftmBaseAddr`)
Disables the FTM peripheral timer overflow interrupt.
- static bool `FTM_HAL_IsOverflowIntEnabled` (uint32_t `baseAddr`)
Reads the bit that controls enabling the FTM timer overflow interrupt.
- static void `FTM_HAL_ClearTimerOverflow` (uint32_t `ftmBaseAddr`)
Clears the timer overflow interrupt flag.
- static bool `FTM_HAL_HasTimerOverflowed` (uint32_t `ftmBaseAddr`)
Returns the FTM peripheral timer overflow interrupt flag.
- static void `FTM_HAL_SetCpwms` (uint32_t `ftmBaseAddr`, uint8_t `mode`)
Sets the FTM center-aligned PWM select.
- static void `FTM_HAL_SetCounter` (uint32_t `ftmBaseAddr`, uint16_t `val`)
Sets the FTM peripheral current counter value.
- static uint16_t `FTM_HAL_GetCounter` (uint32_t `ftmBaseAddr`)
Returns the FTM peripheral current counter value.
- static void `FTM_HAL_SetMod` (uint32_t `ftmBaseAddr`, uint16_t `val`)
Sets the FTM peripheral timer modulo value.
- static uint16_t `FTM_HAL_GetMod` (uint32_t `ftmBaseAddr`)
Returns the FTM peripheral counter modulo value.
- static void `FTM_HAL_SetCounterInitVal` (uint32_t `ftmBaseAddr`, uint16_t `val`)
Sets the FTM peripheral timer counter initial value.
- static uint16_t `FTM_HAL_GetCounterInitVal` (uint32_t `ftmBaseAddr`)
Returns the FTM peripheral counter initial value.
- static void `FTM_HAL_SetChnMSnBAMode` (uint32_t `ftmBaseAddr`, uint8_t `channel`, uint8_t `selection`)
Sets the FTM peripheral timer channel mode.
- static void `FTM_HAL_SetChnEdgeLevel` (uint32_t `ftmBaseAddr`, uint8_t `channel`, uint8_t `level`)
Sets the FTM peripheral timer channel edge level.
- static uint8_t `FTM_HAL_GetChnMode` (uint32_t `ftmBaseAddr`, uint8_t `channel`)
Gets the FTM peripheral timer channel mode.

FlexTimer HAL driver

- static uint8_t **FTM_HAL_GetChnEdgeLevel** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Gets the FTM peripheral timer channel edge level.
- static void **FTM_HAL_SetChnDmaCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*)
Enables or disables the FTM peripheral timer channel DMA.
- static bool **FTM_HAL_IsChnDma** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Returns whether the FTM peripheral timer channel DMA is enabled.
- static void **FTM_HAL_EnableChnInt** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Enables the FTM peripheral timer channel(n) interrupt.
- static void **FTM_HAL_DisableChnInt** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Disables the FTM peripheral timer channel(n) interrupt.
- static bool **FTM_HAL_HasChnEventOccurred** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Returns whether any event for the FTM peripheral timer channel has occurred.
- static void **FTM_HAL_SetChnCountVal** (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint16_t *val*)
Sets the FTM peripheral timer channel counter value.
- static uint16_t **FTM_HAL_GetChnCountVal** (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint16_t *val*)
Gets the FTM peripheral timer channel counter value.
- static uint32_t **FTM_HAL_GetChnEventStatus** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Gets the FTM peripheral timer channel event status.
- static void **FTM_HAL_ClearChnEventStatus** (uint32_t *ftmBaseAddr*, uint8_t *channel*)
Clears the FTM peripheral timer all channel event status.
- static void **FTM_HAL_SetChnOutputMask** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *mask*)
Sets the FTM peripheral timer channel output mask.
- static void **FTM_HAL_SetChnOutputInitState** (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *state*)
Sets the FTM peripheral timer channel output initial state 0 or 1.
- static void **FTM_HAL_SetChnOutputPolarity** (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *pol*)
Sets the FTM peripheral timer channel output polarity.
- static void **FTM_HAL_SetChnFaultInputPolarity** (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *pol*)
Sets the FTM peripheral timer channel input polarity.
- static void **FTM_HAL_EnableFaultInt** (uint32_t *ftmBaseAddr*)
Enables the FTM peripheral timer fault interrupt.
- static void **FTM_HAL_DisableFaultInt** (uint32_t *ftmBaseAddr*)
Disables the FTM peripheral timer fault interrupt.
- static void **FTM_HAL_SetFaultControlMode** (uint32_t *ftmBaseAddr*, uint8_t *mode*)
Defines the FTM fault control mode.
- static void **FTM_HAL_SetCaptureTestCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
Enables or disables the FTM peripheral timer capture test mode.
- static void **FTM_HAL_SetWriteProtectionCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
Enables or disables the FTM write protection.
- static void **FTM_HAL_Enable** (uint32_t *ftmBaseAddr*, bool *enable*)
Enables the FTM peripheral timer group.
- static void **FTM_HAL_SetInitChnOutputCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
Initializes the channels output.
- static void **FTM_HAL_SetPwmSyncMode** (uint32_t *ftmBaseAddr*, bool *enable*)
Sets the FTM peripheral timer sync mode.
- static void **FTM_HAL_SetSoftwareTriggerCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
Enables or disables the FTM peripheral timer software trigger.
- void **FTM_HAL_SetHardwareTrigger** (uint32_t *ftmBaseAddr*, uint8_t *trigger_num*, bool *enable*)
Sets the FTM peripheral timer hardware trigger.
- static void **FTM_HAL_SetOutmaskPwmSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Determines when the OUTMASK register is updated with the value of its buffer.

- static void **FTM_HAL_SetCountReinitSyncCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Determines if the FTM counter is re-initialized when the selected trigger for synchronization is detected.
- static void **FTM_HAL_SetMaxLoadingCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Enables or disables the FTM peripheral timer maximum loading points.
- static void **FTM_HAL_SetMinLoadingCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Enables or disables the FTM peripheral timer minimum loading points.
- uint32_t **FTM_HAL_GetChnPairIndex** (uint8_t *channel*)

Combines the channel control.
- static void **FTM_HAL_SetDualChnFaultCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables the FTM peripheral timer channel pair fault control.
- static void **FTM_HAL_SetDualChnPwmSyncCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables or disables the FTM peripheral timer channel pair counter PWM sync.
- static void **FTM_HAL_SetDualChnDeadtimeCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables or disabled the FTM peripheral timer channel pair deadtime insertion.
- static void **FTM_HAL_SetDualChnDecapCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables or disables the FTM peripheral timer channel dual edge capture decap.
- static void **FTM_HAL_SetDualEdgeCaptureCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables the FTM peripheral timer dual edge capture mode.
- static void **FTM_HAL_SetDualChnCompCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables or disables the FTM peripheral timer channel pair output complement mode.
- static void **FTM_HAL_SetDualChnCombineCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*)

Enables or disables the FTM peripheral timer channel pair output combine mode.
- static void **FTM_HAL_SetDeadtimePrescale** (uint32_t *ftmBaseAddr*, **ftm_deadtime_ps_t** *divider*)

Sets the FTM deadtime divider.
- static void **FTM_HAL_SetDeadtimeCount** (uint32_t *ftmBaseAddr*, uint8_t *count*)

Sets the FTM deadtime value.
- static void **FTM_HAL_SetInitTriggerCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Enables or disables the generation of the trigger when the FTM counter is equal to the CNTIN register.
- void **FTM_HAL_SetChnTriggerCmd** (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*)

Enables or disables the generation of the FTM peripheral timer channel trigger.
- static bool **FTM_HAL_IsChnTriggerGenerated** (uint32_t *ftmBaseAddr*)

Checks whether any channel trigger event has occurred.
- static uint8_t **FTM_HAL_GetDetectedFaultInput** (uint32_t *ftmBaseAddr*)

Gets the FTM detected fault input.
- static bool **FTM_HAL_IsWriteProtectionEnabled** (uint32_t *ftmBaseAddr*)

Checks whether the write protection is enabled.
- static void **FTM_HAL_SetQuadDecoderCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Enables the channel quadrature decoder.
- static void **FTM_HAL_SetQuadPhaseAFilterCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Enables or disables the phase A input filter.
- static void **FTM_HAL_SetQuadPhaseBFilterCmd** (uint32_t *ftmBaseAddr*, bool *enable*)

Enables or disables the phase B input filter.
- static void **FTM_HAL_SetQuadPhaseAPolarity** (uint32_t *ftmBaseAddr*, **ftm_quad_phase_polarity**-

FlexTimer HAL driver

[_t mode](#))

Selects polarity for the quadrature decode phase A input.

- static void [FTM_HAL_SetQuadPhaseBPolarity](#) (uint32_t `ftmBaseAddr`, [ftm_quad_phase_polarity_t mode](#))

Selects polarity for the quadrature decode phase B input.

- static void [FTM_HAL_SetQuadMode](#) (uint32_t `ftmBaseAddr`, [ftm_quad_decode_mode_t quadMode](#))

Sets the encoding mode used in quadrature decoding mode.

- static uint8_t [FTM_HAL_GetQuadDir](#) (uint32_t `ftmBaseAddr`)

Gets the FTM counter direction in quadrature mode.

- static uint8_t [FTM_HAL_GetQuadTimerOverflowDir](#) (uint32_t `ftmBaseAddr`)

Gets the Timer overflow direction in quadrature mode.

- void [FTM_HAL_SetChnInputCaptureFilter](#) (uint32_t `ftmBaseAddr`, uint8_t `channel`, uint8_t `val`)

Sets the FTM peripheral timer channel input capture filter value.

- static void [FTM_HAL_SetFaultInputFilterVal](#) (uint32_t `ftmBaseAddr`, uint32_t `val`)

Sets the fault input filter value.

- static void [FTM_HAL_SetFaultInputFilterCmd](#) (uint32_t `ftmBaseAddr`, uint8_t `inputNum`, bool `val`)

Enables or disables the fault input filter.

- static void [FTM_HAL_SetFaultInputCmd](#) (uint32_t `ftmBaseAddr`, uint8_t `inputNum`, bool `val`)

Enables or disables the fault input.

- static void [FTM_HAL_SetDualChnInvertCmd](#) (uint32_t `ftmBaseAddr`, uint8_t `channel`, bool `val`)

Enables or disables the channel invert for a channel pair.

- static void [FTM_HAL_SetChnSoftwareCtrlCmd](#) (uint32_t `ftmBaseAddr`, uint8_t `channel`, bool `val`)

Enables or disables the channel software output control.

- static void [FTM_HAL_SetChnSoftwareCtrlVal](#) (uint32_t `ftmBaseAddr`, uint8_t `channel`, bool `val`)

Sets the channel software output control value.

- static void [FTM_HAL_SetPwmLoadCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Enables or disables the loading of MOD, CNTIN and CV with values of their write buffer.

- static void [FTM_HAL_SetPwmLoadChnSelCmd](#) (uint32_t `ftmBaseAddr`, uint8_t `channel`, bool `val`)

Includes or excludes the channel in the matching process.

- static void [FTM_HAL_SetGlobalTimeBaseOutputCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Enables or disables the FTM global time base signal generation to other FTM's.

- static void [FTM_HAL_SetGlobalTimeBaseCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Enables or disables the FTM timer global time base.

- static void [FTM_HAL_SetBdmMode](#) (uint32_t `ftmBaseAddr`, uint8_t `val`)

Sets the BDM mode.

- static void [FTM_HAL_SetTofFreq](#) (uint32_t `ftmBaseAddr`, uint8_t `val`)

Sets the FTM timer TOF Frequency.

- static void [FTM_HAL_SetSwoctrlHardwareSyncModeCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Sets the sync mode for the FTM SWOCTRL register when using a hardware trigger.

- static void [FTM_HAL_SetInvctrlHardwareSyncModeCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Sets sync mode for FTM INVCTRL register when using a hardware trigger.

- static void [FTM_HAL_SetOutmaskHardwareSyncModeCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Sets sync mode for FTM OUTMASK register when using a hardware trigger.

- static void [FTM_HAL_SetModCtinCvHardwareSyncModeCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Sets sync mode for FTM MOD, CNTIN and CV registers when using a hardware trigger.

- static void [FTM_HAL_SetCounterHardwareSyncModeCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

Sets sync mode for FTM counter register when using a hardware trigger.

- static void [FTM_HAL_SetSwoctrlSoftwareSyncModeCmd](#) (uint32_t `ftmBaseAddr`, bool `enable`)

- static void **FTM_HAL_SetInvctrlSoftwareSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets sync mode for FTM SWOCTRL register when using a software trigger.*
- static void **FTM_HAL_SetOutmaskSoftwareSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets sync mode for FTM INVCTRL register when using a software trigger.*
- static void **FTM_HAL_SetModCtinCvSoftwareSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets sync mode for FTM OUTMASK register when using a software trigger.*
- static void **FTM_HAL_SetCounterSoftwareSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets sync mode for FTM MOD, CNTIN and CV registers when using a software trigger.*
- static void **FTM_HAL_SetPwmSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets sync mode for FTM counter register when using a software trigger.*
- static void **FTM_HAL_SetSwoctrlPwmSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets the PWM synchronization mode to enhanced or legacy.*
- static void **FTM_HAL_SetInvctrlPwmSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets the SWOCTRL register PWM synchronization mode.*
- static void **FTM_HAL_SetCtinPwmSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets the INVCTRL register PWM synchronization mode.*
- static void **FTM_HAL_SetCntinPwmSyncModeCmd** (uint32_t *ftmBaseAddr*, bool *enable*)
 - Sets the CNTIN register PWM synchronization mode.*
- void **FTM_HAL_Reset** (uint32_t *ftmBaseAddr*, uint32_t *instance*)
 - Resets the FTM registers.*
- void **FTM_HAL_Init** (uint32_t *ftmBaseAddr*)
 - Initializes the FTM.*
- void **FTM_HAL_EnablePwmMode** (uint32_t *ftmBaseAddr*, **ftm_pwm_param_t** **config*, uint8_t *channel*)
 - Enables the FTM timer when it is PWM output mode.*
- void **FTM_HAL_DisablePwmMode** (uint32_t *ftmBaseAddr*, **ftm_pwm_param_t** **config*, uint8_t *channel*)
 - Disables the PWM output mode.*

11.1.1 Data Structure Documentation

11.1.1.1 union **ftm_edge_mode_t**

11.1.1.2 struct **ftm_pwm_param_t**

Data Fields

- **ftm_config_mode_t mode**
 - FlexTimer PWM operation mode.*
- **ftm_pwm_edge_mode_t edgeMode**
 - PWM output mode.*
- **uint32_t uFrequencyHZ**
 - PWM period in Hz.*
- **uint32_t uDutyCyclePercent**
 - PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...*
- **uint16_t uFirstEdgeDelayPercent**
 - Used only in combined PWM mode to generate asymmetrical PWM.*

FlexTimer HAL driver

11.1.1.2.0.27 Field Documentation

11.1.1.2.0.27.1 uint32_t ftm_pwm_param_t::uDutyCyclePercent

100=active signal (100% duty cycle).

11.1.1.2.0.27.2 uint16_t ftm_pwm_param_t::uFirstEdgeDelayPercent

Specifies the delay to the first edge in a PWM period. If unsure please leave as 0, should be specified as percentage of the PWM period

11.1.1.3 struct ftm_phase_params_t

Data Fields

- bool kFtmPhaseInputFilter
 - false: disable phase filter, true: enable phase filter*
- uint32_t kFtmPhaseFilterVal
 - Filter value, used only if phase input filter is enabled.*
- ftm_quad_phase_polarity_t kFtmPhasePolarity
 - kFtmQuadPhaseNormal or kFtmQuadPhaseInvert*

11.1.2 Macro Definition Documentation

11.1.2.1 #define HW_CHAN0 (0U)

11.1.2.2 #define HW_CHAN1 (1U)

11.1.2.3 #define HW_CHAN2 (2U)

11.1.2.4 #define HW_CHAN3 (3U)

11.1.2.5 #define HW_CHAN4 (4U)

11.1.2.6 #define HW_CHAN5 (5U)

11.1.2.7 #define HW_CHAN6 (6U)

11.1.2.8 #define HW_CHAN7 (7U)

11.1.3 Enumeration Type Documentation

11.1.3.1 enum ftm_output_compare_edge_mode_t

Toggle, clear or set.

11.1.4 Function Documentation

11.1.4.1 **static void FTM_HAL_SetClockSource (uint32_t *ftmBaseAddr*,
ftm_clock_source_t *clock*) [inline], [static]**

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>clock</i>	The FTM peripheral clock selection bits - 00: No clock 01: system clock 10: fixed clock 11: External clock

11.1.4.2 static uint8_t FTM_HAL_GetClockSource (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Returns

The FTM clock source selection
bits - 00: No clock 01: system clock 10: fixed clock 11:External clock

11.1.4.3 static void FTM_HAL_SetClockPs (uint32_t *ftmBaseAddr*, *ftm_clock_ps_t ps*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>ps</i>	The FTM peripheral clock pre-scale divider

11.1.4.4 static uint8_t FTM_HAL_GetClockPs (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Returns

The FTM clock pre-scale divider

11.1.4.5 static void FTM_HAL_EnableTimerOverflowInt (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

11.1.4.6 static void FTM_HAL_DisableTimerOverflowInt (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

11.1.4.7 static bool FTM_HAL_IsOverflowIntEnabled (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	FTM module base address.
-----------------	--------------------------

Return values

<i>true</i>	if overflow interrupt is enabled, false if not
-------------	--

11.1.4.8 static void FTM_HAL_ClearTimerOverflow (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

11.1.4.9 static bool FTM_HAL_HasTimerOverflowed (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

FlexTimer HAL driver

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>true</i>	if overflow, false if not
-------------	---------------------------

11.1.4.10 static void FTM_HAL_SetCpwms (uint32_t *ftmBaseAddr*, uint8_t *mode*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>mode</i>	1:upcounting mode 0:up_down counting mode

11.1.4.11 static void FTM_HAL_SetCounter (uint32_t *ftmBaseAddr*, uint16_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>val</i>	FTM timer counter value to be set

11.1.4.12 static uint16_t FTM_HAL_GetCounter (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>current</i>	FTM timer counter value
----------------	-------------------------

11.1.4.13 static void FTM_HAL_SetMod (uint32_t *ftmBaseAddr*, uint16_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>val</i>	The value to be set to the timer modulo

11.1.4.14 static uint16_t FTM_HAL_GetMod (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>FTM</i>	timer modulo value
------------	--------------------

11.1.4.15 static void FTM_HAL_SetCounterInitVal (uint32_t *ftmBaseAddr*, uint16_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>val</i>	initial value to be set

11.1.4.16 static uint16_t FTM_HAL_GetCounterInitVal (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>FTM</i>	timer counter initial value
------------	-----------------------------

11.1.4.17 static void FTM_HAL_SetChnMSnBAMode (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *selection*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>selection</i>	The mode to be set valid value MSnB:MSnA :00,01, 10, 11

11.1.4.18 static void FTM_HAL_SetChnEdgeLevel (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *level*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>level</i>	The rising or falling edge to be set, valid value ELSnB:ELSnA :00,01, 10, 11

11.1.4.19 static uint8_t FTM_HAL_GetChnMode (uint32_t *ftmBaseAddr*, uint8_t *channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

Return values

The MSnB:MSnA mode value, will be 00,01, 10, 11

11.1.4.20 static uint8_t FTM_HAL_GetChnEdgeLevel (uint32_t *ftmBaseAddr*, uint8_t *channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

<i>channel</i>	The FTM peripheral channel number
----------------	-----------------------------------

Return values

<i>The</i>	ELSnB:ELSnA mode value, will be 00,01, 10, 11
------------	---

11.1.4.21 static void FTM_HAL_SetChnDmaCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>val</i>	enable or disable

11.1.4.22 static bool FTM_HAL_IsChnDma (uint32_t *ftmBaseAddr*, uint8_t *channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

Return values

<i>true</i>	if enabled, false if disabled
-------------	-------------------------------

11.1.4.23 static void FTM_HAL_EnableChnInt (uint32_t *ftmBaseAddr*, uint8_t *channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

11.1.4.24 static void FTM_HAL_DisableChnInt (uint32_t *ftmBaseAddr*, uint8_t *channel*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

11.1.4.25 static bool FTM_HAL_HasChnEventOccurred (uint32_t *ftmBaseAddr*, uint8_t *channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

Return values

<i>true</i>	if event occurred, false otherwise.
-------------	-------------------------------------

11.1.4.26 static void FTM_HAL_SetChnCountVal (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint16_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>val</i>	counter value to be set

11.1.4.27 static uint16_t FTM_HAL_GetChnCountVal (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint16_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

Return values

<i>val</i>	return current channel counter value
------------	--------------------------------------

11.1.4.28 static uint32_t FTM_HAL_GetChnEventStatus (*uint32_t ftmBaseAddr, uint8_t channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

Return values

<i>val</i>	return current channel event status value
------------	---

11.1.4.29 static void FTM_HAL_ClearChnEventStatus (*uint32_t ftmBaseAddr, uint8_t channel*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number

Return values

<i>val</i>	return current channel counter value
------------	--------------------------------------

11.1.4.30 static void FTM_HAL_SetChnOutputMask (*uint32_t ftmBaseAddr, uint8_t channel, bool mask*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

FlexTimer HAL driver

<i>channel</i>	The FTM peripheral channel number
<i>mask</i>	mask to be set 0 or 1, unmasked or masked

11.1.4.31 static void FTM_HAL_SetChnOutputInitState (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *state*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>state</i>	counter value to be set 0 or 1

11.1.4.32 static void FTM_HAL_SetChnOutputPolarity (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *pol*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>pol</i>	polarity to be set 0 or 1

11.1.4.33 static void FTM_HAL_SetChnFaultInputPolarity (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *pol*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>pol</i>	polarity to be set, 0: active high, 1:active low

11.1.4.34 static void FTM_HAL_EnableFaultInt (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

11.1.4.35 static void FTM_HAL_DisableFaultInt (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

11.1.4.36 static void FTM_HAL_SetFaultControlMode (uint32_t *ftmBaseAddr*, uint8_t *mode*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>mode,valid</i>	options are 1, 2, 3, 4

11.1.4.37 static void FTM_HAL_SetCaptureTestCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true to enable capture test mode, false to disable

11.1.4.38 static void FTM_HAL_SetWriteProtectionCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

FlexTimer HAL driver

<i>enable</i>	true: Write-protection is enabled, false: Write-protection is disabled
---------------	--

**11.1.4.39 static void FTM_HAL_Enable (uint32_t *ftmBaseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true: all registers including FTM-specific registers are available false: only the TPM-compatible registers are available

11.1.4.40 static void FTM_HAL_SetInitChnOutputCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true: the channels output is initialized according to the state of OUTINIT reg false: has no effect

**11.1.4.41 static void FTM_HAL_SetPwmSyncMode (uint32_t *ftmBaseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true: no restriction both software and hardware triggers can be used false: software trigger can only be used for MOD and CnV synch, hardware trigger only for OUTMASK and FTM counter synch.

11.1.4.42 static void FTM_HAL_SetSoftwareTriggerCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address.
<i>enable</i>	true: software trigger is selected, false: software trigger is not selected

11.1.4.43 void FTM_HAL_SetHardwareTrigger (uint32_t *ftmBaseAddr*, uint8_t *trigger_num*, bool *enable*)

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>trigger_num</i>	0, 1, 2 for trigger0, trigger1 and trigger3
<i>enable</i>	true: enable hardware trigger from field trigger_num for PWM synch false: disable hardware trigger from field trigger_num for PWM synch

11.1.4.44 static void FTM_HAL_SetOutmaskPwmSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true if OUTMASK register is updated only by PWM sync false if OUTMASK register is updated in all rising edges of the system clock

11.1.4.45 static void FTM_HAL_SetCountReinitSyncCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to update FTM counter when triggered , false to count normally

11.1.4.46 static void FTM_HAL_SetMaxLoadingCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to enable maximum loading point, false to disable

11.1.4.47 static void FTM_HAL_SetMinLoadingCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to enable minimum loading point, false to disable

11.1.4.48 uint32_t FTM_HAL_GetChnPairIndex (uint8_t *channel*)

Returns an index for each channel pair.

Parameters

<i>channel</i>	The FTM peripheral channel number.
----------------	------------------------------------

Returns

- 0 for channel pair 0 & 1
- 1 for channel pair 2 & 3
- 2 for channel pair 4 & 5
- 3 for channel pair 6 & 7

11.1.4.49 static void FTM_HAL_SetDualChnFaultCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable fault control, false to disable

11.1.4.50 static void FTM_HAL_SetDualChnPwmSyncCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable PWM synchronization, false to disable

11.1.4.51 static void FTM_HAL_SetDualChnDeadtimeCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable deadtime insertion, false to disable

11.1.4.52 static void FTM_HAL_SetDualChnDecapCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable dual edge capture mode, false to disable

11.1.4.53 static void FTM_HAL_SetDualEdgeCaptureCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable dual edge capture, false to disable

11.1.4.54 static void FTM_HAL_SetDualChnCompCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable complementary mode, false to disable

11.1.4.55 static void FTM_HAL_SetDualChnCombineCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>enable</i>	True to enable channel pair to combine, false to disable

11.1.4.56 static void FTM_HAL_SetDeadtimePrescale (uint32_t *ftmBaseAddr*, ftm_deadtime_ps_t *divider*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>divider</i>	The FTM peripheral prescale divider 0x :divided by 1, 10: divided by 4, 11:divided by 16

11.1.4.57 static void FTM_HAL_SetDeadtimeCount (uint32_t *ftmBaseAddr*, uint8_t *count*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>count</i>	The FTM peripheral prescale divider 0: no counts inserted, 1: 1 count is inserted, 2: 2 count is inserted....

11.1.4.58 static void FTM_HAL_SetInitTriggerCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to enable, false to disable

11.1.4.59 void FTM_HAL_SetChnTriggerCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*)

Enables or disables the when the generation of the FTM peripheral timer channel trigger when the FTM counter is equal to its initial value. Channels 6 and 7 cannot be used as triggers.

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	Channel to be enabled, valid value 0, 1, 2, 3, 4, 5
<i>val</i>	True to enable, false to disable

11.1.4.60 static bool FTM_HAL_IsChnTriggerGenerated (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>true</i>	if there is a channel trigger event, false if not.
-------------	--

11.1.4.61 static uint8_t FTM_HAL_GetDetectedFaultInput (uint32_t *ftmBaseAddr*) [inline], [static]

This function reads the status for all fault inputs

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

FlexTimer HAL driver

Return values

<i>Return</i>	fault byte
---------------	------------

11.1.4.62 static bool FTM_HAL_IsWriteProtectionEnabled (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>True</i>	if enabled, false if not
-------------	--------------------------

11.1.4.63 static void FTM_HAL_SetQuadDecoderCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to enable, false to disable

11.1.4.64 static void FTM_HAL_SetQuadPhaseAFilterCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true enables the phase input filter, false disables the filter

11.1.4.65 static void FTM_HAL_SetQuadPhaseBFilterCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true enables the phase input filter, false disables the filter

11.1.4.66 static void FTM_HAL_SetQuadPhaseAPolarity (uint32_t *ftmBaseAddr*, ftm_quad_phase_polarity_t *mode*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>mode</i>	0: Normal polarity, 1: Inverted polarity

11.1.4.67 static void FTM_HAL_SetQuadPhaseBPolarity (uint32_t *ftmBaseAddr*, ftm_quad_phase_polarity_t *mode*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>mode</i>	0: Normal polarity, 1: Inverted polarity

11.1.4.68 static void FTM_HAL_SetQuadMode (uint32_t *ftmBaseAddr*, ftm_quad_decode_mode_t *quadMode*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>quadMode</i>	0: Phase A and Phase B encoding mode 1: Count and direction encoding mode

11.1.4.69 static uint8_t FTM_HAL_GetQuadDir (uint32_t *ftmBaseAddr*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>I</i>	if counting direction is increasing, 0 if counting direction is decreasing
----------	--

11.1.4.70 static uint8_t FTM_HAL_GetQuadTimerOverflowDir (uint32_t *ftmBaseAddr*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
--------------------	----------------------

Return values

<i>I</i>	if TOF bit was set on the top of counting, 0 if TOF bit was set on the bottom of counting
----------	---

11.1.4.71 void FTM_HAL_SetChnInputCaptureFilter (uint32_t *ftmBaseAddr*, uint8_t *channel*, uint8_t *val*)

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number, only 0,1,2,3, channel 4, 5,6, 7 don't have.
<i>val</i>	Filter value to be set

11.1.4.72 static void FTM_HAL_SetFaultInputFilterVal (uint32_t *ftmBaseAddr*, uint32_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>val</i>	fault input filter value

11.1.4.73 **static void FTM_HAL_SetFaultInputFilterCmd (uint32_t *ftmBaseAddr*, uint8_t *inputNum*, bool *val*) [inline], [static]**

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>inputNum</i>	fault input to be configured, valid value 0, 1, 2, 3
<i>val</i>	true to enable fault input filter, false to disable fault input filter

11.1.4.74 static void FTM_HAL_SetFaultInputCmd (uint32_t *ftmBaseAddr*, uint8_t *inputNum*, bool *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>inputNum</i>	fault input to be configured, valid value 0, 1, 2, 3
<i>val</i>	true to enable fault input, false to disable fault input

11.1.4.75 static void FTM_HAL_SetDualChnInvertCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	The FTM peripheral channel number
<i>val</i>	true to enable channel inverting, false to disable channel inverting

11.1.4.76 static void FTM_HAL_SetChnSoftwareCtrlCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	Channel to be enabled or disabled
<i>val</i>	true to enable, channel output will be affected by software output control false to disable, channel output is unaffected

11.1.4.77 static void FTM_HAL_SetChnSoftwareCtrlVal (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address.
<i>channel</i>	Channel to be configured
<i>val</i>	True to set 1, false to set 0

11.1.4.78 static void FTM_HAL_SetPwmLoadCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true to enable, false to disable

11.1.4.79 static void FTM_HAL_SetPwmLoadChnSelCmd (uint32_t *ftmBaseAddr*, uint8_t *channel*, bool *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>channel</i>	Channel to be configured
<i>val</i>	true means include the channel in the matching process false means do not include channel in the matching process

11.1.4.80 static void FTM_HAL_SetGlobalTimeBaseOutputCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to enable, false to disable

11.1.4.81 static void FTM_HAL_SetGlobalTimeBaseCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	True to enable, false to disable

11.1.4.82 static void FTM_HAL_SetBdmMode (uint32_t *ftmBaseAddr*, uint8_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>val</i>	FTM behaviour in BDM mode, options are 0,1,2,3

11.1.4.83 static void FTM_HAL_SetTofFreq (uint32_t *ftmBaseAddr*, uint8_t *val*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>val</i>	Value of the TOF bit set frequency

11.1.4.84 static void FTM_HAL_SetSwoctrlHardwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means the hardware trigger activates register sync false means the hardware trigger does not activate register sync.

11.1.4.85 static void FTM_HAL_SetInvctrlHardwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means the hardware trigger activates register sync false means the hardware trigger does not activate register sync.

11.1.4.86 static void FTM_HAL_SetOutmaskHardwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means hardware trigger activates register sync false means hardware trigger does not activate register sync.

11.1.4.87 static void FTM_HAL_SetModCntinCvHardwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means hardware trigger activates register sync false means hardware trigger does not activate register sync.

11.1.4.88 static void FTM_HAL_SetCounterHardwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means hardware trigger activates register sync false means hardware trigger does not activate register sync.

11.1.4.89 static void FTM_HAL_SetSwoctrlSoftwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means software trigger activates register sync false means software trigger does not activate register sync.

11.1.4.90 static void FTM_HAL_SetInvctrlSoftwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means software trigger activates register sync false means software trigger does not activate register sync.

11.1.4.91 static void FTM_HAL_SetOutmaskSoftwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means software trigger activates register sync false means software trigger does not activate register sync.

11.1.4.92 static void FTM_HAL_SetModCntinCvSoftwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means software trigger activates register sync false means software trigger does not activate register sync.

11.1.4.93 static void FTM_HAL_SetCounterSoftwareSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means software trigger activates register sync false means software trigger does not activate register sync.

11.1.4.94 static void FTM_HAL_SetPwmSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means use Enhanced PWM synchronization false means to use Legacy mode

11.1.4.95 static void FTM_HAL_SetSwoctrlPwmSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means SWOCTRL register is updated by PWM synch false means SWOCTRL register is updated at all rising edges of system clock

11.1.4.96 static void FTM_HAL_SetInvctrlPwmSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means INVCTRL register is updated by PWM synch false means INVCTRL register is updated at all rising edges of system clock

11.1.4.97 static void FTM_HAL_SetCntinPwmSyncModeCmd (uint32_t *ftmBaseAddr*, bool *enable*) [inline], [static]

FlexTimer HAL driver

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>enable</i>	true means CNTIN register is updated by PWM synch false means CNTIN register is updated at all rising edges of system clock

11.1.4.98 void FTM_HAL_Reset (uint32_t *ftmBaseAddr*, uint32_t *instance*)

Parameters

<i>instance</i>	The FTM instance number
<i>ftmBaseAddr</i>	The FTM base address

11.1.4.99 void FTM_HAL_Init (uint32_t *ftmBaseAddr*)

Parameters

<i>ftmBaseAddr</i>	The FTM base address.
--------------------	-----------------------

11.1.4.100 void FTM_HAL_EnablePwmMode (uint32_t *ftmBaseAddr*, ftm_pwm_param_t * *config*, uint8_t *channel*)

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>config</i>	PWM configuration parameter
<i>channel</i>	The channel or channel pair number(combined mode).

11.1.4.101 void FTM_HAL_DisablePwmMode (uint32_t *ftmBaseAddr*, ftm_pwm_param_t * *config*, uint8_t *channel*)

Parameters

<i>ftmBaseAddr</i>	The FTM base address
<i>config</i>	PWM configuration parameter
<i>channel</i>	The channel or channel pair number(combined mode).

11.2 FlexTimer Peripheral Driver

This chapter describes the programming interface of the FlexTimer Peripheral driver.

Data Structures

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

Functions

- void `FTM_DRV_Init` (uint8_t instance, `ftm_user_config_t` *info)
Initializes the FTM driver.
- void `FTM_DRV_Deinit` (uint8_t instance)
Shuts down the FTM driver.
- void `FTM_DRV_PwmStop` (uint8_t instance, `ftm_pwm_param_t` *param, uint8_t channel)
Stops channel PWM.
- void `FTM_DRV_PwmStart` (uint8_t instance, `ftm_pwm_param_t` *param, uint8_t channel)
Configures duty cycle and frequency and starts outputting PWM on specified channel.
- void `FTM_DRV_QuadDecodeStart` (uint8_t instance, `ftm_phase_params_t` *phaseAParams, `ftm_phase_params_t` *phaseBParams, `ftm_quad_decode_mode_t` quadMode)
Configures the parameters needed and activates quadrature decode mode.
- void `FTM_DRV_QuadDecodeStop` (uint8_t instance)
De-activates quadrature decode mode.
- void `FTM_DRV_CounterStart` (uint8_t instance, `ftm_counting_mode_t` countMode, uint32_t countStartVal, uint32_t countFinalVal, bool enableOverflowInt)
Starts the FTM counter.
- void `FTM_DRV_CounterStop` (uint8_t instance)
Stops the FTM counter.
- uint32_t `FTM_DRV_CounterRead` (uint8_t instance)
Reads back the current value of the FTM counter.
- void `FTM_DRV_SetTimeOverflowIntCmd` (uint32_t instance, bool overflowEnable)
Enables or disables the timer overflow interrupt.
- void `FTM_DRV_SetFaultIntCmd` (uint32_t instance, bool faultEnable)
Enables or disables the fault interrupt.
- void `FTM_DRV_IRQHandler` (uint8_t instance)
Action to take when an FTM interrupt is triggered.

11.2.0.102 FlexCAN Driver

Overview

The FlexTimer module is a timer that supports input capture, output compare, and generation of PWM signals. The current SDK driver only supports generation of PWM signals. The input capture and output compare will be supported in upcoming SDK release.

Initialization

To initialize the FlexTimer driver, call the [FTM_DRV_Init\(\)](#) function and pass the instance number of the FTM you want to use. For instance, to use FTM0, pass a value of 0 to the initialization function. In addition, you should also pass a user configuration structure [ftm_user_config_t](#), as shown here:

```
// FTM configuration structure for user
typedef struct FtmUserConfig {
    uint8_t tofFrequency;
    bool isFTMMode;
    uint8_t BDMMMode;
    bool isWriteProtection;

    bool isTimerOverFlowInterrupt;
    bool isFaultInterrupt;
} ftm_user_config_t;
```

Generate a PWM signal

FTM calls the [FTM_DRV_PwmStart\(\)](#) function to generate a PWM signal. Use this structure to configure different parameters related to the PWM signal.

```
typedef struct FtmPwmParam
{
    ftm_config_mode_t mode;
    ftm_pwm_edge_mode_t edgeMode;
    uint32_t uFrequencyHZ;
    uint32_t uDutyCyclePercent;
    uint16_t uFirstEdgeDelayPercent;
} ftm_pwm_param_t;
```

The mode options are kFtmEdgeAlignedPWM, kFtmCenterAlignedPWM, and kFtmCombinedPWM. For the edge mode, the options available are kFtmHighTrue and kFtmLowTrue. The user should specify the PWM signal frequency in Hz and the duty cycle percentage (value between 0-100). If the PWM mode is kFtmCombinedPWM, the user can choose to specify a value for the uFirstEdgeDelayPercent. This will delay the start of the PWM pulse.

11.2.1 Data Structure Documentation

11.2.1.1 `struct ftm_user_config_t`

11.2.2 Function Documentation

11.2.2.1 `void FTM_DRV_Init(uint8_t instance, ftm_user_config_t * info)`

Parameters

<i>instance</i>	The FTM peripheral instance number.
<i>info</i>	The FTM user configuration structure.

11.2.2.2 void FTM_DRV_Deinit (uint8_t *instance*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
-----------------	-------------------------------------

11.2.2.3 void FTM_DRV_PwmStop (uint8_t *instance*, ftm_pwm_param_t * *param*, uint8_t *channel*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
<i>param</i>	FTM driver PWM parameter to configure PWM options
<i>channel</i>	The channel number. In combined mode, the code will find the channel pair associated with the channel number passed in.

11.2.2.4 void FTM_DRV_PwmStart (uint8_t *instance*, ftm_pwm_param_t * *param*, uint8_t *channel*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
<i>param</i>	FTM driver PWM parameter to configure PWM options
<i>channel</i>	The channel number. In combined mode, the code will find the channel pair associated with the channel number passed in.

11.2.2.5 void FTM_DRV_QuadDecodeStart (uint8_t *instance*, ftm_phase_params_t * *phaseAParams*, ftm_phase_params_t * *phaseBParams*, ftm_quad_decode_mode_t *quadMode*)

FlexTimer Peripheral Driver

Parameters

<i>instance</i>	Instance number of the FTM module.
<i>phaseAParams</i>	Phase A configuration parameters
<i>phaseBParams</i>	Phase B configuration parameters
<i>quadMode</i>	Selects encoding mode used in quadrature decoder mode

11.2.2.6 void FTM_DRV_QuadDecodeStop (uint8_t *instance*)

Parameters

<i>instance</i>	Instance number of the FTM module.
-----------------	------------------------------------

11.2.2.7 void FTM_DRV_CounterStart (uint8_t *instance*, *ftm_counting_mode_t countMode*, uint32_t *countStartVal*, uint32_t *countFinalVal*, bool *enableOverflowInt*)

This function provides access to the FTM counter. The counter can be run in Up-counting and Up-down counting modes. To run the counter in Free running mode, choose Up-counting option and provide 0x0 for the countStartVal and 0xFFFF for countFinalVal.

Parameters

<i>instance</i>	The FTM peripheral instance number.
<i>countMode</i>	The FTM counter mode defined by <i>ftm_counting_mode_t</i> .
<i>countStartVal</i>	The starting value that is stored in the CNTIN register.
<i>countFinalVal</i>	The final value that is stored in the MOD register.
<i>enable-OverflowInt</i>	true: enable timer overflow interrupt; false: disable

11.2.2.8 void FTM_DRV_CounterStop (uint8_t *instance*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
-----------------	-------------------------------------

11.2.2.9 uint32_t FTM_DRV_CounterRead (uint8_t *instance*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
-----------------	-------------------------------------

11.2.2.10 void FTM_DRV_SetTimeOverflowIntCmd (uint32_t *instance*, bool *overflowEnable*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
<i>overflowEnable</i>	true: enable the timer overflow interrupt, false: disable

11.2.2.11 void FTM_DRV_SetFaultIntCmd (uint32_t *instance*, bool *faultEnable*)

Parameters

<i>instance</i>	The FTM peripheral instance number.
<i>faultEnable</i>	true: enable the fault interrupt, false: disable

11.2.2.12 void FTM_DRV_IRQHandler (uint8_t *instance*)

The timer overflow flag is checked and cleared if set.

Parameters

<i>instance</i>	Instance number of the FTM module.
-----------------	------------------------------------

Chapter 12

General Purpose Input/Output (GPIO)

The Kinetis SDK provides both HAL and Peripheral drivers for the General Purpose Input/Output (GPIO) block of Kinetis devices.

Modules

- [GPIO HAL driver](#)

This part describes the programming interface of the GPIO HAL driver.

- [GPIO ISR Definitions](#)

This part describes the programming interface of the GPIO interrupt definitions.

- [GPIO Peripheral Driver](#)

This part describes the programming interface of the GPIO Peripheral driver.

GPIO HAL driver

12.1 GPIO HAL driver

This chapter describes the programming interface of the GPIO HAL driver.

Enumerations

- enum `gpio_pin_direction_t` {
 `kGpioDigitalInput` = 0,
 `kGpioDigitalOutput` = 1 }
 GPIO direction definition.

Configuration

- void `GPIO_HAL_SetPinDir` (uint32_t baseAddr, uint32_t pin, `gpio_pin_direction_t` direction)
 Sets the individual GPIO pin to general input or output.
- static void `GPIO_HAL_SetPortDir` (uint32_t baseAddr, uint32_t direction)
 Sets the GPIO port pins to general input or output.

Status

- static `gpio_pin_direction_t` `GPIO_HAL_GetPinDir` (uint32_t baseAddr, uint32_t pin)
 Gets the current direction of the individual GPIO pin.
- static uint32_t `GPIO_HAL_GetPortDir` (uint32_t baseAddr)
 Gets the GPIO port pins direction.

Output Operation

- void `GPIO_HAL_WritePinOutput` (uint32_t baseAddr, uint32_t pin, uint32_t output)
 Sets the output level of the individual GPIO pin to logic 1 or 0.
- static uint32_t `GPIO_HAL_ReadPinOutput` (uint32_t baseAddr, uint32_t pin)
 Reads the current pin output.
- static void `GPIO_HAL_SetPinOutput` (uint32_t baseAddr, uint32_t pin)
 Sets the output level of the individual GPIO pin to logic 1.
- static void `GPIO_HAL_ClearPinOutput` (uint32_t baseAddr, uint32_t pin)
 Clears the output level of the individual GPIO pin to logic 0.
- static void `GPIO_HAL_TogglePinOutput` (uint32_t baseAddr, uint32_t pin)
 Reverses the current output logic of the individual GPIO pin.
- static void `GPIO_HAL_WritePortOutput` (uint32_t baseAddr, uint32_t portOutput)
 Sets the output of the GPIO port to a specific logic value.
- static uint32_t `GPIO_HAL_ReadPortOutput` (uint32_t baseAddr)
 Reads out all pin output status of the current port.

Input Operation

- static uint32_t `GPIO_HAL_ReadPinInput` (uint32_t baseAddr, uint32_t pin)

- *Reads the current input value of the individual GPIO pin.*
- static uint32_t [GPIO_HAL_ReadPortInput](#) (uint32_t baseAddr)
Reads the current input value of a specific GPIO port.

12.1.0.13 GPIO HAL Driver

Overview

The GPIO HAL driver is designed to access the GPIO hardware registers. It provides sets of API functions for users to configure and control the GPIO ports/pins.

12.1.1 Enumeration Type Documentation

12.1.1.1 enum gpio_pin_direction_t

Enumerator

kGpioDigitalInput Set current pin as digital input.

kGpioDigitalOutput Set current pin as digital output.

12.1.2 Function Documentation

12.1.2.1 void GPIO_HAL_SetPinDir (uint32_t *baseAddr*, uint32_t *pin*, gpio_pin_direction_t *direction*)

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number
<i>direction</i>	GPIO directions <ul style="list-style-type: none"> • kGpioDigitalInput: set to input • kGpioDigitalOutput: set to output

12.1.2.2 static void GPIO_HAL_SetPortDir (uint32_t *baseAddr*, uint32_t *direction*) [inline], [static]

This function operates all 32 port pins.

GPIO HAL driver

Parameters

<i>baseAddr</i>	GPIO base address (HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>direction</i>	GPIO directions <ul style="list-style-type: none">• 0: set to input• 1: set to output• LSB: pin 0• MSB: pin 31

12.1.2.3 static gpio_pin_direction_t GPIO_HAL_GetPinDir (uint32_t *baseAddr*, uint32_t *pin*) [inline], [static]

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number

Returns

GPIO directions

- kGpioDigitalInput: corresponding pin is set to input.
- kGpioDigitalOutput: corresponding pin is set to output.

12.1.2.4 static uint32_t GPIO_HAL_GetPortDir (uint32_t *baseAddr*) [inline], [static]

This function gets all 32-pin directions as a 32-bit integer.

Parameters

<i>baseAddr</i>	GPIO base address (HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
-----------------	--

Returns

GPIO directions. Each bit represents one pin. For each bit:

- 0: corresponding pin is set to input
- 1: corresponding pin is set to output
- LSB: pin 0
- MSB: pin 31

12.1.2.5 **void GPIO_HAL_WritePinOutput (uint32_t *baseAddr*, uint32_t *pin*, uint32_t *output*)**

GPIO HAL driver

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number
<i>output</i>	pin output logic level

**12.1.2.6 static uint32_t GPIO_HAL_ReadPinOutput (uint32_t *baseAddr*, uint32_t *pin*)
[inline], [static]**

Parameters

<i>baseAddr</i>	GPIO base address (HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number

Returns

current pin output status. 0 - Low logic, 1 - High logic

**12.1.2.7 static void GPIO_HAL_SetPinOutput (uint32_t *baseAddr*, uint32_t *pin*)
[inline], [static]**

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number

**12.1.2.8 static void GPIO_HAL_ClearPinOutput (uint32_t *baseAddr*, uint32_t *pin*)
[inline], [static]**

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number

**12.1.2.9 static void GPIO_HAL_TogglePinOutput (uint32_t *baseAddr*, uint32_t *pin*)
[inline], [static]**

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number

12.1.2.10 static void GPIO_HAL_WritePortOutput (uint32_t *baseAddr*, uint32_t *portOutput*) [inline], [static]

This function operates all 32 port pins.

Parameters

<i>baseAddr</i>	GPIO base address (HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>portOutput</i>	<p>data to configure the GPIO output. Each bit represents one pin. For each bit:</p> <ul style="list-style-type: none"> • 0: set logic level 0 to pin • 1: set logic level 1 to pin • LSB: pin 0 • MSB: pin 31

12.1.2.11 static uint32_t GPIO_HAL_ReadPortOutput (uint32_t *baseAddr*) [inline], [static]

This function operates all 32 port pins.

Parameters

<i>baseAddr</i>	GPIO base address (HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
-----------------	--

Returns

current port output status. Each bit represents one pin. For each bit:

- 0: corresponding pin is outputting logic level 0
- 1: corresponding pin is outputting logic level 1
- LSB: pin 0
- MSB: pin 31

12.1.2.12 static uint32_t GPIO_HAL_ReadPinInput (uint32_t *baseAddr*, uint32_t *pin*) [inline], [static]

GPIO HAL driver

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
<i>pin</i>	GPIO port pin number

Returns

GPIO port input value

- 0: Pin logic level is 0, or is not configured for use by digital function.
- 1: Pin logic level is 1

12.1.2.13 **static uint32_t GPIO_HAL_ReadPortInput(uint32_t *baseAddr*) [inline], [static]**

This function gets all 32-pin input as a 32-bit integer.

Parameters

<i>baseAddr</i>	GPIO base address(HW_GPIOA, HW_GPIOB, HW_GPIOC, etc.)
-----------------	---

Returns

GPIO port input data. Each bit represents one pin. For each bit:

- 0: Pin logic level is 0, or is not configured for use by digital function.
- 1: Pin logic level is 1.
- LSB: pin 0
- MSB: pin 31

12.2 GPIO Peripheral Driver

This chapter describes the programming interface of the GPIO Peripheral driver.

Data Structures

- struct `gpio_input_pin_t`
The GPIO input pin configuration structure. [More...](#)
- struct `gpio_output_pin_t`
The GPIO output pin configuration structure. [More...](#)
- struct `gpio_input_pin_user_config_t`
The GPIO input pin structure. [More...](#)
- struct `gpio_output_pin_user_config_t`
The GPIO output pin structure. [More...](#)

GPIO Pin Macros

- #define `GPIO_PINS_OUT_OF_RANGE` (0xFFFFFFFFU)
Indicates the end of a pin configuration structure.
- #define `GPIO_PORT_SHIFT` (0x8U)
Bits shifted for the GPIO port number.
- #define `GPIO_MAKE_PIN`(r, p) (((r)<< `GPIO_PORT_SHIFT`) | (p))
Combines the port number and the pin number into a single scalar value.
- #define `GPIO_EXTRACT_PORT`(v) (((v) >> `GPIO_PORT_SHIFT`) & 0xFFU)
Extracts the port number from a combined port and pin value.
- #define `GPIO_EXTRACT_PIN`(v) ((v) & 0xFFU)
Extracts the pin number from a combined port and pin value.

Initialization

- void `GPIO_DRV_Init` (const `gpio_input_pin_user_config_t` *inputPins, const `gpio_output_pin_user_config_t` *outputPins)
Initialize all GPIO pins used by board.
- void `GPIO_DRV_InputPinInit` (const `gpio_input_pin_user_config_t` *inputPin)
Initializes one GPIO input pin used by board.
- void `GPIO_DRV_OutputPinInit` (const `gpio_output_pin_user_config_t` *outputPin)
Initializes one GPIO output pin used by board.

Pin Direction

- `gpio_pin_direction_t` `GPIO_DRV_GetPinDir` (uint32_t pinName)
Gets the current direction of the individual GPIO pin.
- void `GPIO_DRV_SetPinDir` (uint32_t pinName, `gpio_pin_direction_t` direction)
Sets the current direction of the individual GPIO pin.

Output Operations

- void **GPIO_DRV_WritePinOutput** (uint32_t pinName, uint32_t output)
Sets the output level of the individual GPIO pin to the logic 1 or 0.
- void **GPIO_DRV_SetPinOutput** (uint32_t pinName)
Sets the output level of the individual GPIO pin to the logic 1.
- void **GPIO_DRV_ClearPinOutput** (uint32_t pinName)
Sets the output level of the individual GPIO pin to the logic 0.
- void **GPIO_DRV_TogglePinOutput** (uint32_t pinName)
Reverses current output logic of the individual GPIO pin.

Input Operations

- uint32_t **GPIO_DRV_ReadPinInput** (uint32_t pinName)
Reads the current input value of the individual GPIO pin.

Interrupt

- void **GPIO_DRV_ClearPinIntFlag** (uint32_t pinName)
Clears the individual GPIO pin interrupt status flag.

12.2.0.14 GPIO Peripheral Driver

Overview

The GPIO Peripheral driver configures pins to digital input/output and provides API functions for input/output operations.

GPIO Pin Definitions

The user should define GPIO pins according to the target board configuration and ensure that the definitions are correct. One header file should be defined to save GPIO pin names and the input/output configuration arrays defined in source files.

Include this header file in source files where you want to use GPIO driver to operate GPIO pins.

GPIO pin header file example:

```
// Feel free to change the pin names as what you want
enum _gpio_pins
{
    kGpioLED1 = GPIO_MAKE_PIN(HW_PORTC, 0x0),
    kGpioLED2 = GPIO_MAKE_PIN(HW_PORTC, 0x1),
    kGpioLED3 = GPIO_MAKE_PIN(HW_PORTC, 0x2),
    kGpioLED4 = GPIO_MAKE_PIN(HW_PORTC, 0x3),
};
```

```
// Extern input/output arrays defined in source files.
extern gpio_input_pin_t inputPin[];
extern gpio_output_pin_t outputPin[];
```

Initialization

To initialize the GPIO driver, two arrays, the `gpio_input_pin_t` `inputPin[]` and the `gpio_output_pin_t` `outputPin[]`, should be defined first. Then, call the `GPIO_DRV_Init()` function and pass these two arrays.

Example of the `inputPin` and `outputPin` array definition:

```
#include "gpio/fsl_gpio_driver.h"
#include "gpio_pin_header_file.h"

// Configure kGpioPTA2 as a digital input and enable the interrupt on the rising edge.
gpio_input_pin_t inputPin[] = {
{
    .pinName = kGpioPTA2,
    .config.isPullEnable = false,
    .config.pullSelect = kPortPullDown,
    .config.isPassiveFilterEnabled = false,
    .config.interrupt = kPortIntRisingEdge,
},
{
    //Note: This pinName must be defined here to indicate end of array.
    .pinName = GPIO_PINS_OUT_OF_RANGE,
}
};

// Configure the kGpioLED4 and kGpioPTB9 as a digital output and enable the high drive strength.
gpio_output_pin_t outputPin[] = {
{
    .pinName = kGpioLED4,
    .config.outputLogic = 0,
    .config.slewRate = kPortFastSlewRate,
    .config.driveStrength = kPortHighDriveStrength,
    .config.interrupt = kPortIntDisabled,
},
{
    .pinName = kGpioPTB9,
    .config.outputLogic = 0,
    .config.slewRate = kPortFastSlewRate,
    .config.driveStrength = kPortHighDriveStrength,
    .config.interrupt = kPortIntDisabled,
},
{
    //Note: This pinName must be defined here to indicate the end of array.
    .pinName = GPIO_PINS_OUT_OF_RANGE,
}
};

// Initializes GPIO pins.
GPIO_DRV_Init(inputPin, outputPin);
```

GPIO Peripheral Driver

Note

If the digital filter is enabled, the digital filter clock source and width must be configured before calling the GPIO_DRV_Init function. To configure the digital filter, use this API function from porting the HAL driver. Each pin in the same port shares the same digital filter settings.

```
void PORT_HAL_SetDigitalFilterClock(uint32_t baseAddr, port_digital_filter_clock_source_t clockSource);  
void PORT_HAL_SetDigitalFilterWidth(uint32_t baseAddr, uint8_t width);
```

Output Operations

To use the output operation, configure the target GPIO pin as a digital output in the GPIO_DRV_Init function. The output operations include set, clear, and toggle of the output logic level. Three API functions are provided for these operations:

```
void GPIO_DRV_SetPinOutput(uint32_t pinName);  
void GPIO_DRV_ClearPinOutput(uint32_t pinName);  
void GPIO_DRV_TogglePinOutput(uint32_t pinName);
```

These functions are used when the logic level of the GPIO output is known.

Otherwise, a different function is provided to configure the output logic level according to a passed parameter:

```
void GPIO_DRV_WritePinOutput(uint32_t pinName, uint32_t output);
```

Use this function to change the GPIO output according to the results of other code. Pass an integer as the "uint32_t output" parameter. If that integer is not 0, it generates the high logic. If the integer is 0, it generates the low logic.

Input Operations

To use the input operation, configure the target GPIO pin as a digital input in the GPIO_DRV_Init function. For the input operation, the most commonly used API function is:

```
uint32_t GPIO_DRV_ReadPinInput(uint32_t pinName);
```

This function returns the logic level read from a specific GPIO pin.

If the digital filter is enabled, use this function to disable it:

```
void GPIO_DRV_SetDigitalFilterCmd(uint32_t pinName, bool isDigitalFilterEnabled);
```

Interrupt

Enable a specific pin interrupt in GPIO initialization structures. Both output and input can trigger an interrupt. This API function clears the interrupt flag inside the interrupt handler:

```
void GPIO_DRV_ClearPinIntFlag(uint32_t pinName);
```

12.2.1 Data Structure Documentation

12.2.1.1 struct gpio_input_pin_t

Although every pin is configurable, valid configurations depend on a specific SoC. Users should check the related reference manual to ensure that the specific feature is valid in an individual pin. A configuration of unavailable features is harmless, but takes no effect.

Data Fields

- bool `isPullEnable`
Enable or disable pull.
- `port_pull_t pullSelect`
Select internal pull(up/down) resistor.
- bool `isPassiveFilterEnabled`
Enable or disable passive filter.
- `port_interrupt_config_t interrupt`
Select interrupt/DMA request.

12.2.1.1.0.28 Field Documentation

12.2.1.1.0.28.1 bool gpio_input_pin_t::isPullEnable

12.2.1.1.0.28.2 port_pull_t gpio_input_pin_t::pullSelect

12.2.1.1.0.28.3 bool gpio_input_pin_t::isPassiveFilterEnabled

12.2.1.1.0.28.4 port_interrupt_config_t gpio_input_pin_t::interrupt

12.2.1.2 struct gpio_output_pin_t

Although every pin is configurable, valid configurations depend on a specific SoC. Users should check the related reference manual to ensure that the specific feature is valid in an individual pin. The configuration of unavailable features is harmless, but takes no effect.

Data Fields

- uint32_t `outputLogic`
Set default output logic.
- `port_drive_strength_t driveStrength`
Select fast/slow slew rate.

GPIO Peripheral Driver

12.2.1.2.0.29 Field Documentation

12.2.1.2.0.29.1 `uint32_t gpio_output_pin_t::outputLogic`

12.2.1.2.0.29.2 `port_drive_strength_t gpio_output_pin_t::driveStrength`

Select low/high drive strength.

12.2.1.3 `struct gpio_input_pin_user_config_t`

Although the pinName is defined as a uint32_t type, values assigned to the pinName should be the enumeration names defined in the enum _gpio_pins.

Data Fields

- `uint32_t pinName`
Virtual pin name from enum defined by the user.
- `gpio_input_pin_t config`
Input pin configuration structure.

12.2.1.3.0.30 Field Documentation

12.2.1.3.0.30.1 `uint32_t gpio_input_pin_user_config_t::pinName`

12.2.1.3.0.30.2 `gpio_input_pin_t gpio_input_pin_user_config_t::config`

12.2.1.4 `struct gpio_output_pin_user_config_t`

Although the pinName is defined as a uint32_t type, values assigned to the pinName should be the enumeration names defined in the enum _gpio_pins.

Data Fields

- `uint32_t pinName`
Virtual pin name from enum defined by the user.
- `gpio_output_pin_t config`
Input pin configuration structure.

12.2.1.4.0.31 Field Documentation

12.2.1.4.0.31.1 `uint32_t gpio_output_pin_user_config_t::pinName`

12.2.1.4.0.31.2 `gpio_output_pin_t gpio_output_pin_user_config_t::config`

12.2.2 Macro Definition Documentation

12.2.2.1 `#define GPIO_PINS_OUT_OF_RANGE (0xFFFFFFFFFU)`

12.2.2.2 `#define GPIO_PORT_SHIFT (0x8U)`

12.2.2.3 `#define GPIO_MAKE_PIN(r, p) (((r)<< GPIO_PORT_SHIFT) | (p))`

12.2.2.4 `#define GPIO_EXTRACT_PORT(v) (((v) >> GPIO_PORT_SHIFT) & 0xFFU)`

12.2.2.5 `#define GPIO_EXTRACT_PIN(v) ((v) & 0xFFU)`

12.2.3 Function Documentation

12.2.3.1 `void GPIO_DRV_Init(const gpio_input_pin_user_config_t * inputPins, const gpio_output_pin_user_config_t * outputPins)`

To initialize the GPIO driver, define two arrays similar to the `gpio_input_pin_user_config_t` `inputPin[]` array and the `gpio_output_pin_user_config_t` `outputPin[]` array in the user file. Then, call the `GPIO_DRV_Init()` function and pass in the two arrays. If the input or output pins are not needed, pass in a NULL.

This is an example to define an input pin array:

```
// Configure the kGpioPTA2 as digital input.
gpio_input_pin_user_config_t inputPin[] = {
{
    .pinName = kGpioPTA2,
    .config.isPullEnable = false,
    .config.pullSelect = kPortPullDown,
    .config.isPassiveFilterEnabled = false,
    .config.interrupt = kPortIntDisabled,
},
{
    // Note: This pinName must be defined here to indicate the end of the array.
    .pinName = GPIO_PINS_OUT_OF_RANGE,
}
};
```

Parameters

GPIO Peripheral Driver

<i>inputPins</i>	input GPIO pins pointer.
<i>outputPins</i>	output GPIO pins pointer.

12.2.3.2 void GPIO_DRV_InputPinInit (const gpio_input_pin_user_config_t * *inputPin*)

Parameters

<i>inputPin</i>	input GPIO pins pointer.
-----------------	--------------------------

12.2.3.3 void GPIO_DRV_OutputPinInit (const gpio_output_pin_user_config_t * *outputPin*)

Parameters

<i>outputPin</i>	output GPIO pins pointer.
------------------	---------------------------

12.2.3.4 gpio_pin_direction_t GPIO_DRV_GetPinDir (uint32_t *pinName*)

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
----------------	---

Returns

GPIO directions.

- kGpioDigitalInput: corresponding pin is set as digital input.
- kGpioDigitalOutput: corresponding pin is set as digital output.

12.2.3.5 void GPIO_DRV_SetPinDir (uint32_t *pinName*, gpio_pin_direction_t *direction*)

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
<i>direction</i>	GPIO directions. <ul style="list-style-type: none">• kGpioDigitalInput: corresponding pin is set as digital input.• kGpioDigitalOutput: corresponding pin is set as digital output.

12.2.3.6 void GPIO_DRV_WritePinOutput (uint32_t *pinName*, uint32_t *output*)

GPIO Peripheral Driver

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
<i>output</i>	pin output logic level. <ul style="list-style-type: none">• 0: corresponding pin output low logic level.• Non-0: corresponding pin output high logic level.

12.2.3.7 void GPIO_DRV_SetPinOutput (uint32_t *pinName*)

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
----------------	---

12.2.3.8 void GPIO_DRV_ClearPinOutput (uint32_t *pinName*)

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
----------------	---

12.2.3.9 void GPIO_DRV_TogglePinOutput (uint32_t *pinName*)

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
----------------	---

12.2.3.10 uint32_t GPIO_DRV_ReadPinInput (uint32_t *pinName*)

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
----------------	---

Returns

GPIO port input value.

- 0: Pin logic level is 0, or is not configured for use by digital function.
- 1: Pin logic level is 1.

12.2.3.11 void GPIO_DRV_ClearPinIntFlag (uint32_t *pinName*)

GPIO Peripheral Driver

Parameters

<i>pinName</i>	GPIO pin name defined by the user in the GPIO pin enumeration list.
----------------	---

12.3 GPIO ISR Definitions

This chapter describes the programming interface of the GPIO interrupt definitions.

Chapter 13

Inter-Integrated Circuit (I2C)

The Kinetis SDK provides both HAL and Peripheral drivers for the Inter-Integrated Circuit (I2C) block of Kinetis devices.

Modules

- [I2C Classes](#)

This part describes the C++ classes of the I2C Peripheral driver.

- [I2C HAL driver](#)

This part describes the programming interface of the I2C HAL driver.

- [I2C Master peripheral](#)

This part describes the programming interface of the I2C master mode Peripheral driver.

- [I2C Slave peripheral driver](#)

This part describes the programming interface of the I2C slave mode Peripheral driver.

I2C HAL driver

13.1 I2C HAL driver

This chapter describes the programming interface of the I2C HAL driver.

Enumerations

- enum `i2c_status_t` { ,
 `kStatus_I2C_Busy` = 0x3U,
 `kStatus_I2C_Timeout` = 0x4U,
 `kStatus_I2C_ReceivedNak` = 0x5U,
 `kStatus_I2C_SlaveTxUnderrun` = 0x6U,
 `kStatus_I2C_SlaveRxOverrun` = 0x7U,
 `kStatus_I2C_AribtrationLost` = 0x8U }

I2C status return codes.

- enum `i2c_status_flag_t`

I2C status flags.

- enum `i2c_direction_t` {
 `kI2CReceive` = 0U,
 `kI2CSend` = 1U }

Direction of master and slave transfers.

Module controls

- void `I2C_HAL_Init` (uint32_t baseAddr)
Restores the I2C peripheral to reset state.
- static void `I2C_HAL_Enable` (uint32_t baseAddr)
Enables the I2C module operation.
- static void `I2C_HAL_Disable` (uint32_t baseAddr)
Disables the I2C module operation.

DMA

- static void `I2C_HAL_SetDmaCmd` (uint32_t baseAddr, bool enable)
Enables or disables the DMA support.
- static bool `I2C_HAL_GetDmaCmd` (uint32_t baseAddr)
Returns whether I2C DMA support is enabled.

Pin functions

- static void `I2C_HAL_SetHighDriveCmd` (uint32_t baseAddr, bool enable)
Controls the drive capability of the I2C pads.
- static void `I2C_HAL_SetGlitchWidth` (uint32_t baseAddr, uint8_t glitchWidth)
Controls the width of the programmable glitch filter.

Low power

- static void [I2C_HAL_SetWakeupCmd](#) (uint32_t baseAddr, bool enable)
Controls the I2C wakeup enable.

Baud rate

- [i2c_status_t I2C_HAL_SetBaudRate](#) (uint32_t baseAddr, uint32_t sourceClockInHz, uint32_t kbps, uint32_t *absoluteError_Hz)
Sets the I2C bus frequency for master transactions.
- static void [I2C_HAL_SetFreqDiv](#) (uint32_t baseAddr, uint8_t mult, uint8_t icr)
Sets the I2C baud rate multiplier and table entry.
- static void [I2C_HAL_SetSlaveBaudCtrlCmd](#) (uint32_t baseAddr, bool enable)
Slave baud rate control.

Bus operations

- void [I2C_HAL_SendStart](#) (uint32_t baseAddr)
Sends a START or a Repeated START signal on the I2C bus.
- static void [I2C_HAL_SendStop](#) (uint32_t baseAddr)
Sends a STOP signal on the I2C bus.
- static void [I2C_HAL_SendAck](#) (uint32_t baseAddr)
Causes an ACK to be sent on the bus.
- static void [I2C_HAL_SendNak](#) (uint32_t baseAddr)
Causes a NAK to be sent on the bus.
- static void [I2C_HAL_SetDirMode](#) (uint32_t baseAddr, [i2c_direction_t](#) direction)
Selects either transmit or receive mode.
- static [i2c_direction_t I2C_HAL_GetDirMode](#) (uint32_t baseAddr)
Returns the currently selected transmit or receive mode.

Data transfer

- static uint8_t [I2C_HAL_ReadByte](#) (uint32_t baseAddr)
Returns the last byte of data read from the bus and initiate another read.
- static void [I2C_HAL_WriteByte](#) (uint32_t baseAddr, uint8_t byte)
Writes one byte of data to the I2C bus.

Slave address

- void [I2C_HAL_SetAddress7bit](#) (uint32_t baseAddr, uint8_t address)
Sets the primary 7-bit slave address.
- void [I2C_HAL_SetAddress10bit](#) (uint32_t baseAddr, uint16_t address)
Sets the primary slave address and enables 10-bit address mode.
- static void [I2C_HAL_SetExtensionAddrCmd](#) (uint32_t baseAddr, bool enable)
Enables or disables the extension address (10-bit).

I2C HAL driver

- static bool [I2C_HAL_GetExtensionAddrCmd](#) (uint32_t baseAddr)
Returns whether the extension address is enabled or not.
- static void [I2C_HAL_SetGeneralCallCmd](#) (uint32_t baseAddr, bool enable)
Controls whether the general call address is recognized.
- static void [I2C_HAL_SetRangeMatchCmd](#) (uint32_t baseAddr, bool enable)
Enables or disables the slave address range matching.
- static void [I2C_HAL_SetUpperAddress7bit](#) (uint32_t baseAddr, uint8_t address)
Sets the upper slave address.

Status

- static bool [I2C_HAL_GetStatusFlag](#) (uint32_t baseAddr, [i2c_status_flag_t](#) statusFlag)
Gets the I2C status flag state.
- static bool [I2C_HAL_IsMaster](#) (uint32_t baseAddr)
Returns whether the I2C module is in master mode.
- static void [I2C_HAL_ClearArbitrationLost](#) (uint32_t baseAddr)
Clears the arbitration lost flag.

Interrupt

- static void [I2C_HAL_SetIntCmd](#) (uint32_t baseAddr, bool enable)
Enables or disables I2C interrupt requests.
- static bool [I2C_HAL.GetIntCmd](#) (uint32_t baseAddr)
Returns whether the I2C interrupts are enabled.
- static bool [I2C_HAL_IsIntPending](#) (uint32_t baseAddr)
Returns the current I2C interrupt flag.
- static void [I2C_HAL_ClearInt](#) (uint32_t baseAddr)
Clears the I2C interrupt if set.

13.1.0.12 I2C HAL Driver

ICR Table

ICR (hex)	SCL divider	SDA hold value	SCL start hold value	SCL stop hold value
00	20	7	6	11
01	22	7	7	12
02	24	8	8	13
03	26	8	9	14
04	28	9	10	15
05	30	9	11	16
06	34	10	13	18
07	40	10	16	21

Clock rate formulas

I2C baud rate = bus_clock_Hz / (mult * SCL_divider)

SDA hold time = bus_clock_period_s * mult * SDA_hold_value

SCL start hold time = bus_clock_period_s * mult * SCL_start_hold_value

SCL stop hold time = bus_clock_period_s * mult * SCL_stop_hold_value

13.1.1 Enumeration Type Documentation

13.1.1.1 enum i2c_status_t

Enumerator

kStatusI2C_Busy The master is already performing a transfer.

kStatusI2C_Timeout The transfer timed out.

kStatusI2C_ReceivedNak The slave device sent a NAK in response to a byte.

kStatusI2C_SlaveTxUnderrun I2C Slave TX Underrun error.

kStatusI2C_SlaveRxOverrun I2C Slave RX Overrun error.

kStatusI2C_ArbitrationLost I2C Arbitration Lost error.

13.1.1.2 enum i2c_status_flag_t

13.1.1.3 enum i2c_direction_t

Enumerator

kI2CReceive Master and slave receive.

kI2CSend Master and slave transmit.

13.1.2 Function Documentation

13.1.2.1 void I2C_HAL_Init(uint32_t baseAddr)

Parameters

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

13.1.2.2 static void I2C_HAL_Enable(uint32_t baseAddr) [inline], [static]

I2C HAL driver

Parameters

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

13.1.2.3 static void I2C_HAL_Disable(uint32_t *baseAddr*) [inline], [static]

Parameters

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

13.1.2.4 static void I2C_HAL_SetDmaCmd(uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>enable</i>	Pass true to enable DMA transfer signalling

13.1.2.5 static bool I2C_HAL_GetDmaCmd(uint32_t *baseAddr*) [inline], [static]

Parameters

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

Return values

<i>true</i>	I2C DMA is enabled.
<i>false</i>	I2C DMA is disabled.

13.1.2.6 static void I2C_HAL_SetHighDriveCmd(uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>enable</i>	Passing true will enable high drive mode of the I2C pads. False sets normal drive mode.

13.1.2.7 static void I2C_HAL_SetGlitchWidth (uint32_t *baseAddr*, uint8_t *glitchWidth*) [inline], [static]

Controls the width of the glitch, in terms of bus clock cycles, that the filter must absorb. The filter does not allow any glitch whose size is less than or equal to this width setting, to pass.

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>glitchWidth</i>	Maximum width in bus clock cycles of the glitches that is filtered. Pass zero to disable the glitch filter.

13.1.2.8 static void I2C_HAL_SetWakeupCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

The I2C module can wake the MCU from low power mode with no peripheral bus running when slave address matching occurs.

Parameters

<i>baseAddr</i>	The I2C peripheral base address.
<i>enable</i>	true - Enables the wakeup function in low power mode. false - Normal operation. No interrupt is generated when address matching in low power mode.

I2C HAL driver

13.1.2.9 i2c_status_t I2C_HAL_SetBaudRate (uint32_t *baseAddr*, uint32_t *sourceClockInHz*, uint32_t *kbps*, uint32_t * *absoluteError_Hz*)

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>sourceClockInHz</i>	I2C source input clock in Hertz
<i>kbps</i>	Requested bus frequency in kilohertz. Common values are either 100 or 400.
<i>absoluteError_Hz</i>	If this parameter is not NULL, it is filled in with the difference in Hertz between the requested bus frequency and the closest frequency possible given available divider values.

Return values

<i>kStatus_Success</i>	The baud rate was changed successfully. However, there is no guarantee on the minimum error. If you want to ensure that the baud was set to within a certain error, then use the <i>absoluteError_Hz</i> parameter.
<i>kStatus_OutOfRange</i>	The requested baud rate was not within the range of rates supported by the peripheral.

13.1.2.10 static void I2C_HAL_SetFreqDiv (uint32_t *baseAddr*, uint8_t *mult*, uint8_t *icr*) [inline], [static]

Use this function to set the I2C bus frequency register values directly, if they are known in advance.

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>mult</i>	Value of the MULT bitfield, ranging from 0-2.
<i>icr</i>	The ICR bitfield value, which is the index into an internal table in the I2C hardware that selects the baud rate divisor and SCL hold time.

13.1.2.11 static void I2C_HAL_SetSlaveBaudCtrlCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Enables an independent slave mode baud rate at the maximum frequency. This forces clock stretching on the SCL in very fast I2C modes.

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>enable</i>	true - Slave baud rate is independent of the master baud rate; false - The slave baud rate follows the master baud rate and clock stretching may occur.

13.1.2.12 void I2C_HAL_SendStart (uint32_t *baseAddr*)

This function is used to initiate a new master mode transfer by sending the START signal. It is also used to send a Repeated START signal when a transfer is already in progress.

Parameters

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

13.1.2.13 static void I2C_HAL_SendStop (uint32_t *baseAddr*) [inline], [static]

This function changes the direction to receive.

Parameters

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

13.1.2.14 static void I2C_HAL_SendAck (uint32_t *baseAddr*) [inline], [static]

This function specifies that an ACK signal is sent in response to the next received byte.

Note that the behavior of this function is changed when the I2C peripheral is placed in Fast ACK mode. In this case, this function causes an ACK signal to be sent in response to the current byte, rather than the next received byte.

Parameters

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

13.1.2.15 static void I2C_HAL_SendNak (uint32_t *baseAddr*) [inline], [static]

This function specifies that a NAK signal is sent in response to the next received byte.

Note that the behavior of this function is changed when the I2C peripheral is placed in the Fast ACK mode. In this case, this function causes an NAK signal to be sent in response to the current byte, rather than the

I2C HAL driver

next received byte.

Parameters

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

13.1.2.16 static void I2C_HAL_SetDirMode (uint32_t *baseAddr*, i2c_direction_t *direction*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address.
<i>direction</i>	Specifies either transmit mode or receive mode. The valid values are: <ul style="list-style-type: none"> • #kI2CTransmit • kI2CReceive

13.1.2.17 static i2c_direction_t I2C_HAL_GetDirMode (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Return values

#kI2CTransmit	I2C is configured for master or slave transmit mode.
kI2CReceive	I2C is configured for master or slave receive mode.

13.1.2.18 static uint8_t I2C_HAL_ReadByte (uint32_t *baseAddr*) [inline], [static]

In a master receive mode, calling this function initiates receiving the next byte of data.

Parameters

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

Returns

This function returns the last byte received while the I2C module is configured in master receive or slave receive mode.

I2C HAL driver

13.1.2.19 static void I2C_HAL_WriteByte (uint32_t *baseAddr*, uint8_t *byte*) [inline], [static]

When this function is called in the master transmit mode, a data transfer is initiated. In slave mode, the same function is available after an address match occurs.

In a master transmit mode, the first byte of data written following the start bit or repeated start bit is used for the address transfer and must consist of the slave address (in bits 7-1) concatenated with the required R/#W bit (in position bit 0).

Parameters

<i>baseAddr</i>	The I2C peripheral base address.
<i>byte</i>	The byte of data to transmit.

13.1.2.20 void I2C_HAL_SetAddress7bit (uint32_t *baseAddr*, uint8_t *address*)

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>address</i>	The slave address in the upper 7 bits. Bit 0 of this value must be 0.

13.1.2.21 void I2C_HAL_SetAddress10bit (uint32_t *baseAddr*, uint16_t *address*)

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>address</i>	The 10-bit slave address, in bits [10:1] of the value. Bit 0 must be 0.

13.1.2.22 static void I2C_HAL_SetExtensionAddrCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>enable</i>	true: 10-bit address is enabled. false: 10-bit address is not enabled.

13.1.2.23 static bool I2C_HAL_GetExtensionAddrCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Returns

true: 10-bit address is enabled. false: 10-bit address is not enabled.

13.1.2.24 static void I2C_HAL_SetGeneralCallCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>enable</i>	Whether to enable the general call address.

13.1.2.25 static void I2C_HAL_SetRangeMatchCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address.
<i>enable</i>	Pass true to enable range address matching. You must also call I2C_HAL_SetUpperAddress7bit() to set the upper address.

13.1.2.26 static void I2C_HAL_SetUpperAddress7bit (uint32_t *baseAddr*, uint8_t *address*) [inline], [static]

This slave address is used as a secondary slave address. If range address matching is enabled, this slave address acts as the upper bound on the slave address range.

This function sets only a 7-bit slave address. If 10-bit addressing was enabled by calling [I2C_HAL_SetAddress10bit\(\)](#), then the top 3 bits set with that function are also used with the address set with this function to form a 10-bit address.

Passing 0 for the *address* parameter disables matching the upper slave address.

I2C HAL driver

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>address</i>	The upper slave address in the upper 7 bits. Bit 0 of this value must be 0. In addition, this address must be greater than the primary slave address that is set by calling I2C_HAL_SetAddress7bit() .

13.1.2.27 static bool I2C_HAL_GetStatusFlag (uint32_t *baseAddr*, i2c_status_flag_t *statusFlag*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address.
<i>statusFlag</i>	The status flag, defined in type i2c_status_flag_t.

Returns

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

- true: related status flag is being set.
- false: related status flag is not set.

13.1.2.28 static bool I2C_HAL_IsMaster (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Return values

<i>true</i>	The module is in master mode, which implies it is also performing a transfer.
<i>false</i>	The module is in slave mode.

13.1.2.29 static void I2C_HAL_ClearArbitrationLost (uint32_t *baseAddr*) [inline], [static]

Parameters

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

13.1.2.30 static void I2C_HAL_SetIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The I2C peripheral base address
<i>enable</i>	Pass true to enable interrupt, flase to disable.

13.1.2.31 static bool I2C_HAL_GetIntCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Return values

<i>true</i>	I2C interrupts are enabled.
<i>false</i>	I2C interrupts are disabled.

13.1.2.32 static bool I2C_HAL_IsIntPending (uint32_t *baseAddr*) [inline], [static]

Parameters

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

Return values

<i>true</i>	An interrupt is pending.
<i>false</i>	No interrupt is pending.

13.1.2.33 static void I2C_HAL_ClearInt (uint32_t *baseAddr*) [inline], [static]

I2C HAL driver

Parameters

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

13.2 I2C Slave peripheral driver

This chapter describes the programming interface of the I2C slave mode Peripheral driver.

Data Structures

- struct [i2c_slave_state_t](#)
Definition of application-implemented callback functions used by the I2C slave driver. [More...](#)

I2C Slave

- void [I2C_DRV_SlaveInit](#) (uint32_t instance, uint8_t address, [i2c_slave_state_t](#) *slave)
Initializes the I2C module.
- void [I2C_DRV_SlaveDeinit](#) (uint32_t instance)
Shuts down the I2C slave driver.
- [i2c_slave_source_t](#) [I2C_DRV_SlaveInstallDataSource](#) (uint32_t instance, [i2c_slave_source_t](#) function)
Installs or uninstalls slave data source callback function.
- [i2c_slave_sink_t](#) [I2C_DRV_SlaveInstallDataSink](#) (uint32_t instance, [i2c_slave_sink_t](#) function)
Installs or uninstalls slave data sink callback function.

13.2.1 Data Structure Documentation

13.2.1.1 struct i2c_slave_state_t

Data Fields

- [i2c_slave_source_t](#) [dataSource](#)
Callback to get byte to transmit.
- [i2c_slave_sink_t](#) [dataSink](#)
Callback to put received byte.

13.2.1.1.0.32 Field Documentation

13.2.1.1.0.32.1 [i2c_slave_source_t](#) [i2c_slave_state_t::dataSource](#)

13.2.1.1.0.32.2 [i2c_slave_sink_t](#) [i2c_slave_state_t::dataSink](#)

13.2.2 Function Documentation

13.2.2.1 void [I2C_DRV_SlaveInit](#) ([uint32_t](#) *instance*, [uint8_t](#) *address*, [i2c_slave_state_t](#) **slave*)

Saves the application callback info, turns on the clock to the module, enables the device, and enables interrupts. Sets the I2C to slave mode. IOMUX should be handled in the init_hardware() function.

I2C Slave peripheral driver

Parameters

<i>instance</i>	Instance number of the I2C module.
<i>address</i>	7-bit address for slave.
<i>slave</i>	Pointer of the slave run-time structure.

13.2.2.2 void I2C_DRV_SlaveDeinit (*uint32_t instance*)

Clears the control register and turns off the clock to the module.

Parameters

<i>instance</i>	Instance number of the I2C module.
-----------------	------------------------------------

13.2.2.3 i2c_slave_source_t I2C_DRV_SlaveInstallDataSource (*uint32_t instance*, *i2c_slave_source_t function*)

Parameters

<i>instance</i>	Instance number of the I2C module.
<i>function</i>	Function to be installed. Passing NULL uninstalls the callback.

Returns

Callback function installed before current installation.

13.2.2.4 i2c_slave_sink_t I2C_DRV_SlaveInstallDataSink (*uint32_t instance*, *i2c_slave_sink_t function*)

Parameters

<i>instance</i>	Instance number of the I2C module.
<i>function</i>	Function to be installed. Passing NULL uninstalls the callback.

Returns

Callback function installed before installation.

13.3 I2C Master peripheral

This chapter describes the programming interface of the I2C master mode Peripheral driver.

Data Structures

- struct [i2c_device_t](#)
Information necessary to communicate with an I2C slave device. [More...](#)
- struct [i2c_master_state_t](#)
Internal driver state information. [More...](#)

I2C Master

- void [I2C_DRV_MasterInit](#) (uint32_t instance, [i2c_master_state_t](#) *master)
Initializes the I2C master mode driver.
- void [I2C_DRV_MasterDeinit](#) (uint32_t instance)
Shuts down the driver.
- void [I2C_DRV_MasterSetBaudRate](#) (uint32_t instance, const [i2c_device_t](#) *device)
Configures the I2C bus to access a device.
- [i2c_status_t](#) [I2C_DRV_MasterSendDataBlocking](#) (uint32_t instance, const [i2c_device_t](#) *device, uint8_t *cmdBuff, uint32_t cmdSize, uint8_t *txBuff, uint32_t txSize, uint32_t timeout_ms)
Performs a blocking send transaction on the I2C bus.
- [i2c_status_t](#) [I2C_DRV_MasterReceiveDataBlocking](#) (uint32_t instance, const [i2c_device_t](#) *device, uint8_t *cmdBuff, uint32_t cmdSize, uint8_t *rxBuff, uint32_t rxSize, uint32_t timeout_ms)
Performs a blocking receive transaction on the I2C bus.

13.3.0.5 I2C Master Driver

Overview

The I2C master Peripheral driver provides functions for a master device to send and receive data.

Initialization

To initialize the I2C driver, an [i2c_master_state_t](#) type variable should be defined by user and pass in [i2c_init](#). There is no need to assign value to that variable. The I2C driver only needs that part of memory. Because all I2C master API functions need the run-time structure, it should be maintained as long as the driver is used.

Because I2C drivers use the OSA_Delay form OSA layer, [OSA_Init](#) must be called before calling I2C transaction API functions. Otherwise, the API functions do not work.

I2C Master peripheral

Data Transactions

I2C master driver mainly provides two APIs for data transactions:

I2C_DRV_MasterSendDataBlocking

I2C_DRV_MasterReceiveDataBlocking

Both of these two functions will perform a blocking transaction, that means: the function will not return till all data are sent/received OR time out happens.

Before calling these API functions, the application should define the slave device with "i2c_device_t" type. Note that, if the slave is a 10-bit address, the first 6 bits must be 011110 in binary.

```
typedef struct I2CDevice
{
    uint16_t address; /* Slave's 7-bit or 10-bit address. If 10-bit address, the first 6 bits must be
                       011110 in binary.*/
    uint32_t baudRate_kbps; /* The baud rate in kbps to use with this slave device.*/
} i2c_device_t;
```

If the buadRate_kbps inside the `i2c_device_t` object is changed, it is automatically applied to the next send/receive transaction.

13.3.1 Data Structure Documentation

13.3.1.1 struct i2c_device_t

Data Fields

- `uint16_t address`
Slave's 7-bit or 10-bit address.
- `uint32_t baudRate_kbps`
The baud rate in kbps to use with this slave device.

13.3.1.1.0.33 Field Documentation

13.3.1.1.0.33.1 `uint16_t i2c_device_t::address`

If 10-bit address, the first 6 bits must be 011110 in binary.

13.3.1.1.0.33.2 `uint32_t i2c_device_t::baudRate_kbps`

13.3.1.2 struct i2c_master_state_t

Note

The contents of this structure are internal to the driver and should not be modified by users. Also, contents of the structure are subject to change in future releases.

13.3.2 Function Documentation

13.3.2.1 `void I2C_DRV_MasterInit (uint32_t instance, i2c_master_state_t * master)`

I2C Master peripheral

Parameters

<i>instance</i>	The I2C peripheral instance number.
<i>master</i>	The pointer to the I2C master driver state structure.

13.3.2.2 void I2C_DRV_MasterDeinit (uint32_t *instance*)

Parameters

<i>instance</i>	The I2C peripheral instance number.
-----------------	-------------------------------------

13.3.2.3 void I2C_DRV_MasterSetBaudRate (uint32_t *instance*, const i2c_device_t * *device*)

Parameters

<i>instance</i>	The I2C peripheral instance number.
<i>device</i>	The pointer to the I2C device information structure.

13.3.2.4 i2c_status_t I2C_DRV_MasterSendDataBlocking (uint32_t *instance*, const i2c_device_t * *device*, uint8_t * *cmdBuff*, uint32_t *cmdSize*, uint8_t * *txBuff*, uint32_t *txSize*, uint32_t *timeout_ms*)

Parameters

<i>instance</i>	The I2C peripheral instance number.
<i>device</i>	The pointer to the I2C device information structure.
<i>cmdBuff</i>	The pointer to the commands to be transferred.
<i>cmdSize</i>	The length in bytes of the commands to be transferred.
<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>timeout_ms</i>	A timeout for the transfer in microseconds.
-------------------	---

Returns

Error or success status returned by API.

13.3.2.5 **i2c_status_t I2C_DRV_MasterReceiveDataBlocking (uint32_t *instance*, const i2c_device_t * *device*, uint8_t * *cmdBuff*, uint32_t *cmdSize*, uint8_t * *rxBuff*, uint32_t *rxSize*, uint32_t *timeout_ms*)**

Parameters

<i>instance</i>	The I2C peripheral instance number.
<i>device</i>	The pointer to the I2C device information structure.
<i>cmdBuff</i>	The pointer to the commands to be transferred.
<i>cmdSize</i>	The length in bytes of the commands to be transferred.
<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>timeout_ms</i>	A timeout for the transfer in microseconds.

Returns

Error or success status returned by API.

I2C Classes

13.4 I2C Classes

This chapter describes the C++ classes of the I2C Peripheral driver.

Chapter 14

Low Power Timer (LPTMR)

The Kinetis SDK provides both HAL and Peripheral drivers for the Low Power Timer (LPTMR) block of Kinetis devices.

Modules

- [LPTMR HAL driver](#)
This part describes the programming interface of the LPTMR HAL driver.
- [LPTMR Peripheral Driver](#)
This part describes the programming interface of the LPTMR Peripheral driver.

14.1 LPTMR HAL driver

This chapter describes the programming interface of the LPTMR HAL driver.

Enumerations

- enum `lptmr_pin_select_t` {
 `kLptmrPinSelectCmpOut` = 0x0U,
 `kLptmrPinSelectLptmrAlt1` = 0x1U,
 `kLptmrPinSelectLptmrAlt2` = 0x2U,
 `kLptmrPinSelectLptmrAlt3` = 0x3U }
 LPTMR pin selection.
- enum `lptmr_pin_polarity_t` {
 `kLptmrPinPolarityActiveHigh` = 0x0U,
 `kLptmrPinPolarityActiveLow` = 0x1U }
 LPTMR pin polarity, used while in pulse counter mode.
- enum `lptmr_timer_mode_t` {
 `kLptmrTimerModeTimeCounter` = 0x0U,
 `kLptmrTimerModePluseCounter` = 0x1U }
 LPTMR timer mode selection.
- enum `lptmr_prescaler_value_t` {
 `kLptmrPrescalerDivide2` = 0x0U,
 `kLptmrPrescalerDivide4GlichFiltch2` = 0x1U,
 `kLptmrPrescalerDivide8GlichFiltch4` = 0x2U,
 `kLptmrPrescalerDivide16GlichFiltch8` = 0x3U,
 `kLptmrPrescalerDivide32GlichFiltch16` = 0x4U,
 `kLptmrPrescalerDivide64GlichFiltch32` = 0x5U,
 `kLptmrPrescalerDivide128GlichFiltch64` = 0x6U,
 `kLptmrPrescalerDivide256GlichFiltch128` = 0x7U,
 `kLptmrPrescalerDivide512GlichFiltch256` = 0x8U,
 `kLptmrPrescalerDivide1024GlichFiltch512` = 0x9U,
 `kLptmrPrescalerDivide2048GlichFiltch1024` = 0xAU,
 `kLptmrPrescalerDivide4096GlichFiltch2048` = 0xBU,
 `kLptmrPrescalerDivide8192GlichFiltch4096` = 0xCU,
 `kLptmrPrescalerDivide16384GlichFiltch8192` = 0xDU,
 `kLptmrPrescalerDivide32768GlichFiltch16384` = 0xEU,
 `kLptmrPrescalerDivide65535GlichFiltch32768` = 0xFU }
 LPTMR prescaler value.
- enum `lptmr_prescaler_clock_source_t` {
 `kLptmrPrescalerClockSourceMcgIrcClk` = 0x0U,
 `kLptmrPrescalerClockSourceLpo` = 0x1U,
 `kLptmrPrescalerClockSourceErClk32K` = 0x2U,
 `kLptmrPrescalerClockSourceOscErClk` = 0x3U }
 LPTMR clock source selection.
- enum `lptmr_status_t` {

```

kStatus_LPTMR_Success = 0x0U,
kStatus_LPTMR_NotInitialized = 0x1U,
kStatus_LPTMR_NullArgument = 0x2U,
kStatus_LPTMR_InvalidPrescalerValue = 0x3U,
kStatus_LPTMR_InvalidInTimeCounterMode = 0x4U,
kStatus_LPTMR_InvalidInPluseCounterMode = 0x5U,
kStatus_LPTMR_InvalidPlusePeriodCount = 0x6U,
kStatus_LPTMR_TcfNotSet = 0x7U,
kStatus_LPTMR_TimerPeriodUsTooSmall = 0x8U,
kStatus_LPTMR_TimerPeriodUsTooLarge = 0x9U }
```

LPTMR status return codes.

LPTMR HAL.

- static void [LPTMR_HAL_Enable](#) (uint32_t baseAddr)
Enables the LPTMR module operation.
- static void [LPTMR_HAL_Disable](#) (uint32_t baseAddr)
Disables the LPTMR module operation.
- static bool [LPTMR_HAL_IsEnabled](#) (uint32_t baseAddr)
Checks whether the LPTMR module is enabled.
- static void [LPTMR_HAL_ClearIntFlag](#) (uint32_t baseAddr)
Clears the LPTMR interrupt flag if set.
- static bool [LPTMR_HAL_IsIntPending](#) (uint32_t baseAddr)
Returns the current LPTMR interrupt flag.
- static void [LPTMR_HAL_SetIntCmd](#) (uint32_t baseAddr, bool enable)
Enables or disables the LPTMR interrupt.
- static bool [LPTMR_HAL_GetIntCmd](#) (uint32_t baseAddr)
Returns whether the LPTMR interrupt is enabled.
- static void [LPTMR_HAL_SetPinSelectMode](#) (uint32_t baseAddr, [lptmr_pin_select_t](#) pinSelect)
Selects the LPTMR pulse input pin select.
- static [lptmr_pin_select_t](#) [LPTMR_HAL_GetPinSelectMode](#) (uint32_t baseAddr)
Returns the LPTMR pulse input pin select.
- static void [LPTMR_HAL_SetPinPolarityMode](#) (uint32_t baseAddr, [lptmr_pin_polarity_t](#) pinPolarity)
Selects the LPTMR pulse input pin polarity.
- static [lptmr_pin_polarity_t](#) [LPTMR_HAL_GetPinPolarityMode](#) (uint32_t baseAddr)
Returns the LPTMR pulse input pin polarity.
- static void [LPTMR_HAL_SetFreeRunningCmd](#) (uint32_t baseAddr, bool enable)
Enables or disables the LPTMR free running.
- static bool [LPTMR_HAL_GetFreeRunningCmd](#) (uint32_t baseAddr)
Returns whether the LPTMR free running is enabled.
- static void [LPTMR_HAL_SetTimerModeMode](#) (uint32_t baseAddr, [lptmr_timer_mode_t](#) timerMode)
Selects the LPTMR working mode.
- static [lptmr_timer_mode_t](#) [LPTMR_HAL_GetTimerModeMode](#) (uint32_t baseAddr)
Returns the LPTMR working mode.
- static void [LPTMR_HAL_SetPrescalerValueMode](#) (uint32_t baseAddr, [lptmr_prescaler_value_t](#) prescaleValue)

LPTMR HAL driver

- static `lptmr_prescaler_value_t LPTMR_HAL_GetPrescalerValueMode` (uint32_t baseAddr)
Selects the LPTMR prescaler value.
- static void `LPTMR_HAL_SetPrescalerCmd` (uint32_t baseAddr, bool enable)
Returns the LPTMR prescaler value.
- static bool `LPTMR_HAL_GetPrescalerCmd` (uint32_t baseAddr)
Enables or disables the LPTMR prescaler.
- static void `LPTMR_HAL_SetPrescalerClockSourceMode` (uint32_t baseAddr, `lptmr_prescaler_clock_source_t` prescalerClockSource)
Returns whether the LPTMR prescaler is enabled.
- static `lptmr_prescaler_clock_source_t LPTMR_HAL_GetPrescalerClockSourceMode` (uint32_t baseAddr)
Selects the LPTMR clock source.
- static
`lptmr_prescaler_clock_source_t LPTMR_HAL_SetCompareValue` (uint32_t baseAddr, uint32_t compareValue)
Gets the LPTMR clock source.
- static void `LPTMR_HAL_SetCompareValue` (uint32_t baseAddr, uint32_t compareValue)
Sets the LPTMR compare value.
- static uint32_t `LPTMR_HAL_GetCompareValue` (uint32_t baseAddr)
Gets the LPTMR compare value.
- static uint32_t `LPTMR_HAL_GetCounterValue` (uint32_t baseAddr)
Gets the LPTMR counter value.
- void `LPTMR_HAL_Init` (uint32_t baseAddr)
Restores the LPTMR module to reset state.

14.1.1 Enumeration Type Documentation

14.1.1.1 enum lptmr_pin_select_t

Enumerator

- kLptmrPinSelectCmpOut*** Lptmr Pin is CMP0 output pin.
- kLptmrPinSelectLptmrAlt1*** Lptmr Pin is LPTMR_ALT1 pin.
- kLptmrPinSelectLptmrAlt2*** Lptmr Pin is LPTMR_ALT2 pin.
- kLptmrPinSelectLptmrAlt3*** Lptmr Pin is LPTMR_ALT3 pin.

14.1.1.2 enum lptmr_pin_polarity_t

Enumerator

- kLptmrPinPolarityActiveHigh*** Pulse Counter input source is active-high.
- kLptmrPinPolarityActiveLow*** Pulse Counter input source is active-low.

14.1.1.3 enum lptmr_timer_mode_t

Enumerator

- kLptmrTimerModeTimeCounter*** Time Counter mode.

kLptmrTimerModePluseCounter Pulse Counter mode.

14.1.1.4 enum lptmr_prescaler_value_t

Enumerator

kLptmrPrescalerDivide2 Prescaler divide 2, glitch filter invalid.
kLptmrPrescalerDivide4GlichFiltch2 Prescaler divide 4, glitch filter 2.
kLptmrPrescalerDivide8GlichFiltch4 Prescaler divide 8, glitch filter 4.
kLptmrPrescalerDivide16GlichFiltch8 Prescaler divide 16, glitch filter 8.
kLptmrPrescalerDivide32GlichFiltch16 Prescaler divide 32, glitch filter 16.
kLptmrPrescalerDivide64GlichFiltch32 Prescaler divide 64, glitch filter 32.
kLptmrPrescalerDivide128GlichFiltch64 Prescaler divide 128, glitch filter 64.
kLptmrPrescalerDivide256GlichFiltch128 Prescaler divide 256, glitch filter 128.
kLptmrPrescalerDivide512GlichFiltch256 Prescaler divide 512, glitch filter 256.
kLptmrPrescalerDivide1024GlichFiltch512 Prescaler divide 1024, glitch filter 512.
kLptmrPrescalerDivide2048GlichFiltch1024 Prescaler divide 2048 glitch filter 1024.
kLptmrPrescalerDivide4096GlichFiltch2048 Prescaler divide 4096, glitch filter 2048.
kLptmrPrescalerDivide8192GlichFiltch4096 Prescaler divide 8192, glitch filter 4096.
kLptmrPrescalerDivide16384GlichFiltch8192 Prescaler divide 16384, glitch filter 8192.
kLptmrPrescalerDivide32768GlichFiltch16384 Prescaler divide 32768, glitch filter 16384.
kLptmrPrescalerDivide65535GlichFiltch32768 Prescaler divide 65535, glitch filter 32768.

14.1.1.5 enum lptmr_prescaler_clock_source_t

Enumerator

kLptmrPrescalerClockSourceMcgIrcClk Clock source is MCGIRCLK.
kLptmrPrescalerClockSourceLpo Clock source is LPO.
kLptmrPrescalerClockSourceErClk32K Clock source is ERCLK32K.
kLptmrPrescalerClockSourceOscErClk Clock source is OSCERCLK.

14.1.1.6 enum lptmr_status_t

Enumerator

kStatus_LPTMR_Success Succeed.
kStatus_LPTMR_NotInitialized LPTMR is not initialized yet.
kStatus_LPTMR_NullArgument Argument is NULL.
kStatus_LPTMR_InvalidPrescalerValue Value 0 is not valid in pulse counter mode.
kStatus_LPTMR_InvalidInTimeCounterMode Function can not called in time counter mode.
kStatus_LPTMR_InvalidInPluseCounterMode Function can not called in pulse counter mode.

LPTMR HAL driver

kStatus_LPTMR_InvalidPulsePeriodCount Pulse period count must be integer multiples of the glitch filter divider.

kStatus_LPTMR_TcfNotSet If LPTMR is enabled, compare register can only altered when TCF is set.

kStatus_LPTMR_TimerPeriodUsTooSmall Timer period time is too small for current clock source.

kStatus_LPTMR_TimerPeriodUsTooLarge Timer period time is too large for current clock source.

14.1.2 Function Documentation

14.1.2.1 static void LPTMR_HAL_Enable(uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

14.1.2.2 static void LPTMR_HAL_Disable(uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

14.1.2.3 static bool LPTMR_HAL_IsEnabled(uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Return values

<i>true</i>	LPTMR module is enabled.
<i>false</i>	LPTMR module is disabled.

14.1.2.4 static void LPTMR_HAL_ClearIntFlag(uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

14.1.2.5 static bool LPTMR_HAL_IsIntPending (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address
-----------------	-----------------------------------

Return values

<i>true</i>	An interrupt is pending.
<i>false</i>	No interrupt is pending.

14.1.2.6 static void LPTMR_HAL_SetIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address
<i>enable</i>	Pass true to enable LPTMR interrupt

14.1.2.7 static bool LPTMR_HAL_GetIntCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Return values

<i>true</i>	LPTMR interrupt is enabled.
-------------	-----------------------------

LPTMR HAL driver

<i>false</i>	LPTMR interrupt is disabled.
--------------	------------------------------

14.1.2.8 static void LPTMR_HAL_SetPinSelectMode (uint32_t *baseAddr*, lptmr_pin_select_t *pinSelect*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
<i>pinSelect</i>	Specifies LPTMR pulse input pin select, see lptmr_pin_select_t

14.1.2.9 static lptmr_pin_select_t LPTMR_HAL_GetPinSelectMode (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

LPTMR pulse input pin select, see [lptmr_pin_select_t](#)

14.1.2.10 static void LPTMR_HAL_SetPinPolarityMode (uint32_t *baseAddr*, lptmr_pin_polarity_t *pinPolarity*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
<i>pinPolarity</i>	Specifies LPTMR pulse input pin polarity, see lptmr_pin_polarity_t

14.1.2.11 static lptmr_pin_polarity_t LPTMR_HAL_GetPinPolarityMode (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

LPTMR pulse input pin polarity, see [lptmr_pin_polarity_t](#)

14.1.2.12 static void LPTMR_HAL_SetFreeRunningCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address
<i>enable</i>	Pass true to enable LPTMR free running

14.1.2.13 static bool LPTMR_HAL_GetFreeRunningCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Return values

<i>true</i>	LPTMR free running is enabled.
<i>false</i>	LPTMR free running is disabled.

14.1.2.14 static void LPTMR_HAL_SetTimerModeMode (uint32_t *baseAddr*, lptmr_timer_mode_t *timerMode*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

LPTMR HAL driver

<i>timerMode</i>	Specifies LPTMR working mode, see lptmr_timer_mode_t
------------------	--

14.1.2.15 static lptmr_timer_mode_t LPTMR_HAL_GetTimerModeMode (uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

LPTMR working mode, see [lptmr_timer_mode_t](#)

14.1.2.16 static void LPTMR_HAL_SetPrescalerValueMode (uint32_t baseAddr, lptmr_prescaler_value_t prescaleValue) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
<i>prescaleValue</i>	Specifies LPTMR prescaler value, see lptmr_prescaler_value_t

14.1.2.17 static lptmr_prescaler_value_t LPTMR_HAL_GetPrescalerValueMode (uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

LPTMR prescaler value, see [lptmr_prescaler_value_t](#)

14.1.2.18 static void LPTMR_HAL_SetPrescalerCmd (uint32_t baseAddr, bool enable) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address
<i>enable</i>	Pass true to enable LPTMR free running

14.1.2.19 static bool LPTMR_HAL_GetPrescalerCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Return values

<i>true</i>	LPTMR prescaler is enabled.
<i>false</i>	LPTMR prescaler is disabled.

14.1.2.20 static void LPTMR_HAL_SetPrescalerClockSourceMode (uint32_t *baseAddr*, lptmr_prescaler_clock_source_t *prescalerClockSource*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
<i>prescaler-ClockSource</i>	Specifies LPTMR clock source, see lptmr_prescaler_clock_source_t

14.1.2.21 static lptmr_prescaler_clock_source_t LPTMR_HAL_GetPrescalerClockSource-Mode (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

LPTMR clock source, see [lptmr_prescaler_clock_source_t](#)

14.1.2.22 **static void LPTMR_HAL_SetCompareValue (uint32_t *baseAddr*, uint32_t *compareValue*) [inline], [static]**

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
<i>compareValue</i>	Specifies LPTMR compare value, less than 0xFFFFU

**14.1.2.23 static uint32_t LPTMR_HAL_GetCompareValue (uint32_t *baseAddr*)
[inline], [static]**

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

Current LPTMR compare value

**14.1.2.24 static uint32_t LPTMR_HAL_GetCounterValue (uint32_t *baseAddr*)
[inline], [static]**

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address.
-----------------	------------------------------------

Returns

Current LPTMR counter value

14.1.2.25 void LPTMR_HAL_Init (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	The LPTMR peripheral base address
-----------------	-----------------------------------

LPTMR Peripheral Driver

14.2 LPTMR Peripheral Driver

This chapter describes the programming interface of the LPTMR Peripheral driver.

Data Structures

- struct [lptmr_user_config_t](#)
Data structure to initialize the LPTMR. [More...](#)
- struct [lptmr_state_t](#)
Internal driver state information. [More...](#)

TypeDefs

- [typedef void\(* lptmr_callback_t \)\(void\)](#)
Defines a type of the user-defined callback function.

LPTMR Driver

- [lptmr_status_t LPTMR_DRV_Init \(uint32_t instance, const lptmr_user_config_t *userConfigPtr, lptmr_state_t *userStatePtr\)](#)
Initializes the LPTMR driver.
- [lptmr_status_t LPTMR_DRV_Deinit \(uint32_t instance\)](#)
De-initializes the LPTMR driver.
- [lptmr_status_t LPTMR_DRV_Start \(uint32_t instance\)](#)
Starts the LPTMR counter.
- [lptmr_status_t LPTMR_DRV_Stop \(uint32_t instance\)](#)
Stops the LPTMR counter.
- [lptmr_status_t LPTMR_DRV_SetTimerPeriodUs \(uint32_t instance, uint32_t us\)](#)
Configures the LPTMR timer period in microseconds.
- [uint32_t LPTMR_DRV_GetCurrentTimeUs \(uint32_t instance\)](#)
Gets the current LPTMR time in microseconds.
- [lptmr_status_t LPTMR_DRV_SetPlusePeriodCount \(uint32_t instance, uint32_t plusePeriodCount\)](#)
Sets the pulse period value.
- [uint32_t LPTMR_DRV_GetCurrentPluseCount \(uint32_t instance\)](#)
Gets the current pulse count.
- [lptmr_status_t LPTMR_DRV_InstallCallback \(uint32_t instance, lptmr_callback_t userCallback\)](#)
Installs the user-defined callback in the LPTMR module.
- [void LPTMR_DRV_IRQHandler \(uint32_t instance\)](#)
Driver-defined ISR in the LPTMR module.

14.2.0.26 LPTMR Peripheral Driver

Overview

The LPTMR driver is used to configure the LPTMR.

Initialization

To initialize the LPTMR module, call the [LPTMR_DRV_Init\(\)](#) function and pass in the user configuration data structure. This function automatically enables the LPTMR module and clock .

After calling the [LPTMR_DRV_Init\(\)](#) function is called, call the LPTRM_DRV_Start to start the LPTMR. To stop the LPTMR counter, call the [LPTMR_DRV_Stop](#).

This is example code to configure the driver:

```
/* Defines the LPTM user configuration structure . */
const lptmr_user_config_t config =
{
    .timerMode = kLptmrTimerModeTimerCounter, /* timer counter is selected*/
    .freeRunningEnable = false, /* free running is disabled */
    .intEnable = true, /* interrupt is enabled */
    .compareValue = 1024, /* compare value is 1024 */
    .prescalerEnable = true, /* prescaler is enabled */
    .prescalerClockSource = kLptmrPrescalerClockSourceLpo, /* prescaler clock
        is LPO */
    .prescalerValue = kLptmrPrescalerDivide2, /* prescaler divide is 2 */
};

/* Initializes the LPTMR 0. */
LPTMR_DRV_Init(0,&config);

/* Starts the LPTMR 0. */
LPTMR_DRV_Start(0);

/* Stops LPTMR 0. */
LPTMR_DRV_Stop(0);

/* Deinitializes LPTMR 0. */
LPTMR_DRV_Deinit(0);
```

LPTMR Interrupt

1.Enable the LPTMR interrupt. The LPTMR interrupt is enabled in the user configuration structure with the lptmr_user_config_t.intEnable=true.

2.Define the LPTMR IRQ function.

```
void LPTimer_IRQHandler()
{
    if(true == LPTMR_DRV_IsIntPending(0))
    {
        LPTMR_DRV_ClearIntFlag(0);
    }
    lptmrIsrAssertCount++;
}
```

14.2.1 Data Structure Documentation

14.2.1.1 struct lptmr_user_config_t

This structure is used when initializing the LPTMR during the LPTMR_DRV_Init function call.

LPTMR Peripheral Driver

Data Fields

- `lptmr_timer_mode_t timerMode`
Timer counter mode or pulse counter mode.
- `lptmr_pin_select_t pinSelect`
LPTMR pulse input pin select.
- `lptmr_pin_polarity_t pinPolarity`
LPTMR pulse input pin polarity.
- `bool freeRunningEnable`
Free running configure.
- `bool prescalerEnable`
Prescaler enable configure.
- `lptmr_prescaler_clock_source_t prescalerClockSource`
LPTMR clock source.
- `lptmr_prescaler_value_t prescalerValue`
Prescaler value.
- `bool isInterruptEnabled`
Timer interrupt 0-disable/1-enable.

14.2.1.1.0.34 Field Documentation

14.2.1.1.0.34.1 `bool lptmr_user_config_t::freeRunningEnable`

True means enable free running

14.2.1.1.0.34.2 `bool lptmr_user_config_t::prescalerEnable`

True means enable prescaler

14.2.1.2 `struct lptmr_state_t`

The contents of this structure are internal to the driver and should not be modified by users. Contents of the structure are subject to change in future releases.

Data Fields

- `lptmr_callback_t userCallbackFunc`
Callback function that is executed in ISR.

14.2.1.2.0.35 Field Documentation

14.2.1.2.0.35.1 lptmr_callback_t lptmr_state_t::userCallbackFunc

14.2.2 Function Documentation

14.2.2.1 lptmr_status_t LPTMR_DRV_Init (uint32_t *instance*, const lptmr_user_config_t * *userConfigPtr*, lptmr_state_t * *userStatePtr*)

This function initializes the LPTMR. The LPTMR can be initialized as a time counter or pulse counter, which is determined by the timerMode in the [lptmr_user_config_t](#). pinSelect and pinPolarity do not need to be configured while working as a time counter.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
<i>userConfigPtr</i>	The pointer to the LPTMR user configure structure, see lptmr_user_config_t .
<i>userStatePtr</i>	The pointer to the structure of the context memory, see lptmr_state_t .

Returns

kStatus_LPTMR_Success means succeed, otherwise means failed.

14.2.2.2 lptmr_status_t LPTMR_DRV_Deinit (uint32_t *instance*)

This function de-initializes the LPTMR. It disables the interrupt and turns off the LPTMR clock.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
-----------------	---------------------------------------

Returns

kStatus_LPTMR_Success means succeed, otherwise means failed.

14.2.2.3 lptmr_status_t LPTMR_DRV_Start (uint32_t *instance*)

This function starts the LPTMR counter. Ensure that all necessary configurations are set before calling this function.

LPTMR Peripheral Driver

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
-----------------	---------------------------------------

Returns

kStatus_LPTMR_Success means success. Otherwise, means failure.

14.2.2.4 lptmr_status_t LPTMR_DRV_Stop (uint32_t *instance*)

This function stops the LPTMR counter.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
-----------------	---------------------------------------

Returns

kStatus_LPTMR_Success means success. Otherwise, means failure.

14.2.2.5 lptmr_status_t LPTMR_DRV_SetTimerPeriodUs (uint32_t *instance*, uint32_t *us*)

This function configures the LPTMR time period while the LPTMR is working as a time counter. After the time period in microseconds, the callback function is called. This function cannot be called while the LPTMR is working as a pulse counter. The value in microseconds (μ s) should be integer multiple of the clock source time slice. If the clock source is 1 kHz, then both 2000 μ s and 3000 μ s are valid while 2500 μ s gets the same result as the 2000 μ s, because 2500 μ s cannot be generated in 1 kHz clock source.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
<i>us</i>	time period in microseconds.

Returns

kStatus_LPTMR_Success means success. Otherwise, means failure.

14.2.2.6 uint32_t LPTMR_DRV_GetCurrentTimeUs (uint32_t *instance*)

This function gets the current time while working as a time counter. This function cannot be called while working as a pulse counter.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
-----------------	---------------------------------------

Returns

current time in microsecond unit.

14.2.2.7 lptmr_status_t LPTMR_DRV_SetPlusePeriodCount (uint32_t *instance*, uint32_t *plusePeriodCount*)

This function configures the pulse period of the LPTMR while working as a pulse counter. After the count of plusePeriodValue pulse is captured, the callback function is called. The period value must be in integer multiples of the glitch filter divider. This function cannot be called while working as a time counter.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
<i>plusePeriod-Count</i>	pulse period value.

Returns

kStatus_LPTMR_Success means success. Otherwise, means failure.

14.2.2.8 uint32_t LPTMR_DRV_GetCurrentPluseCount (uint32_t *instance*)

This function gets the current pulse count captured on the pulse input pin. This function cannot be called while working as a time counter.

Parameters

<i>instance</i>	The LPTMR peripheral instance number.
-----------------	---------------------------------------

Returns

pulse count captured on the pulse input pin.

14.2.2.9 lptmr_status_t LPTMR_DRV_InstallCallback (uint32_t *instance*, lptmr_callback_t *userCallback*)

This function installs the user-defined callback in the LPTMR module. When an LPTMR interrupt request is served, the callback is executed inside the ISR.

LPTMR Peripheral Driver

Parameters

<i>instance</i>	LPTMR instance ID.
<i>userCallback</i>	User-defined callback function.

Returns

kStatus_LPTMR_Success means success. Otherwise, means failure.

14.2.2.10 void LPTMR_DRV_IRQHandler (uint32_t *instance*)

This function is the driver-defined ISR in LPTMR module. It includes the process for interrupt mode defined by driver. Currently, it is called inside the system-defined ISR.

Parameters

<i>instance</i>	LPTMR instance ID.
-----------------	--------------------

Chapter 15

Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

The Kinetis SDK provides both HAL and Peripheral drivers for the Low Power Universal Asynchronous Receiver/Transmitter (LPUART) block of Kinetis devices.

Modules

- [LPUART HAL driver](#)
This part describes the programming interface of the LPUART HAL driver.
- [LPUART Types Definitions](#)
This part describes the LPUART type definitions.
- [LPUART peripheral Driver](#)
This part describes the programming interface of the LPUART Peripheral driver.

15.1 LPUART HAL driver

This chapter describes the programming interface of the LPUART HAL driver.

Data Structures

- struct [lpuart_idle_line_config_t](#)
Structure for idle line configuration settings. [More...](#)

Enumerations

- enum [lpuart_status_t](#) {
 kStatus_LPUART_BaudRateCalculationError,
 kStatus_LPUART_BaudRatePercentDiffExceeded,
 kStatus_LPUART_BitCountNotSupported,
 kStatus_LPUART_StopBitCountNotSupported,
 kStatus_LPUART_RxStandbyModeError,
 kStatus_LPUART_ClearStatusFlagError,
 kStatus_LPUART_MSBFirstNotSupported,
 kStatus_LPUART_Resync_NotSupported,
 kStatus_LPUART_TxNotDisabled,
 kStatus_LPUART_RxNotDisabled,
 kStatus_LPUART_TxOrRxNotDisabled,
 kStatus_LPUART_TxBusy,
 kStatus_LPUART_RxBusy,
 kStatus_LPUART_NoTransmitInProgress,
 kStatus_LPUART_NoReceiveInProgress,
 kStatus_LPUART_InvalidInstanceNumber,
 kStatus_LPUART_InvalidBitSetting,
 kStatus_LPUART_OverSamplingNotSupported,
 kStatus_LPUART_BothEdgeNotSupported,
 kStatus_LPUART_Timeout }
 Error codes for the LPUART driver.
- enum [lpuart_stop_bit_count_t](#) {
 kLpuartOneStopBit = 0,
 kLpuartTwoStopBit = 1 }
 LPUART number of stop bits.
- enum [lpuart_parity_mode_t](#) {
 kLpuartParityDisabled = 0x0,
 kLpuartParityEven = 0x2,
 kLpuartParityOdd = 0x3 }
 LPUART parity mode.
- enum [lpuart_bit_count_per_char_t](#) {
 kLpuart8BitsPerChar = 0,
 kLpuart9BitsPerChar = 1,

- `kLpuart10BitsPerChar = 2 }`
LPUART number of bits in a character.
- enum `lpuart_operation_config_t` {

`kLpuartOperates = 0,`

`kLpuartStops = 1 }`
LPUART operation configuration constants.
- enum `lpuart_wakeup_method_t` {

`kLpuartIdleLineWake = 0,`

`kLpuartAddrMarkWake = 1 }`
LPUART wakeup from standby method constants.
- enum `lpuart_idle_line_select_t` {

`kLpuartIdleLineAfterStartBit = 0,`

`kLpuartIdleLineAfterStopBit = 1 }`
LPUART idle line detect selection types.
- enum `lpuart_break_char_length_t` {

`kLpuartBreakChar10BitMinimum = 0,`

`kLpuartBreakChar13BitMinimum = 1 }`
LPUART break character length settings for transmit/detect.
- enum `lpuart_singlewire_txdir_t` {

`kLpuartSinglewireTxdirIn = 0,`

`kLpuartSinglewireTxdirOut = 1 }`
LPUART single-wire mode TX direction.
- enum `lpuart_match_config_t` {

`kLpuartAddressMatchWakeup = 0,`

`kLpuartIdleMatchWakeup = 1,`

`kLpuartMatchOnAndMatchOff = 2,`

`kLpuartEnablesRwuOnDataMatch = 3 }`
LPUART Configures the match addressing mode used.
- enum `lpuart_ir_tx_pulsewidth_t` {

`kLpuartIrThreeSixteenthsWidth = 0,`

`kLpuartIrOneSixteenthWidth = 1,`

`kLpuartIrOneThirtysecondsWidth = 2,`

`kLpuartIrOneFourthWidth = 3 }`
LPUART infra-red transmitter pulse width options.
- enum `lpuart_idle_config_t` {

`kLpuart_1_IdleChar = 0,`

`kLpuart_2_IdleChar = 1,`

`kLpuart_4_IdleChar = 2,`

`kLpuart_8_IdleChar = 3,`

`kLpuart_16_IdleChar = 4,`

`kLpuart_32_IdleChar = 5,`

`kLpuart_64_IdleChar = 6,`

`kLpuart_128_IdleChar = 7 }`
LPUART Configures the number of idle characters that must be received before the IDLE flag is set.
- enum `lpuart_cts_source_t` {

`kLpuartCtsSourcePin = 0,`

`kLpuartCtsSourceInvertedReceiverMatch = 1 }`

LPUART HAL driver

LPUART Transmits the CTS Configuration.

- enum `lpuart_cts_config_t` {
 `kLpuartCtsSampledOnEachCharacter` = 0,
 `kLpuartCtsSampledOnIdle` = 1 }

LPUART Transmits CTS Source. Configures if the CTS state is checked at the start of each character or only when the transmitter is idle.

- enum `lpuart_status_flag_t` {
 `kLpuartTxDataRegEmpty` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_TDRE`,
 `kLpuartTxComplete` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_TC`,
 `kLpuartRxDataRegFull` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_RDRF`,
 `kLpuartIdleLineDetect` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_IDLE`,
 `kLpuartRxOverrun` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_OR`,
 `kLpuartNoiseDetect` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_NF`,
 `kLpuartFrameErr` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_FE`,
 `kLpuartParityErr` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_PF`,
 `kLpuartLineBreakDetect` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_LBKDE`,
 `kLpuartRxActiveEdgeDetect` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_RXEDGIF`,
 `kLpuartRxActive` = `LPUART_STAT_REG_ID << LPUART_SHIFT | BP_LPUART_STAT_RAF`
}

LPUART status flags.

- enum `lpuart_interrupt_t` {
 `kLpuartIntLinBreakDetect` = `LPUART_BAUD_REG_ID << LPUART_SHIFT | BP_LPUART_BAUD_LBKDE`,
 `kLpuartIntRxActiveEdge` = `LPUART_BAUD_REG_ID << LPUART_SHIFT | BP_LPUART_BAUD_RXEDGIE`,
 `kLpuartIntTxDataRegEmpty` = `LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CTRL_TIE`,
 `kLpuartIntTxComplete` = `LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CTRL_TCIE`,
 `kLpuartIntRxDataRegFull` = `LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CTRL_RIE`,
 `kLpuartIntIdleLine` = `LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CTRL_IIE`,
 `kLpuartIntRxOverrun` = `LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CTRL_ORIE`,
 `kLpuartIntNoiseErrFlag` = `LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CT-`

```

    RL_NEIE,
    kLpuartIntFrameErrFlag = LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CT-
    RL_FEIE,
    kLpuartIntParityErrFlag = LPUART_CTRL_REG_ID << LPUART_SHIFT | BP_LPUART_CT-
    RL_PEIE }

```

LPUART interrupt configuration structure, default settings are 0 (disabled)

LPUART Common Configurations

- void [LPUART_HAL_Init](#) (uint32_t baseAddr)
Initializes the LPUART controller to known state.
- void [LPUART_HAL_EnableTransmitter](#) (uint32_t baseAddr)
Enables the LPUART transmitter.
- static void [LPUART_HAL_DisableTransmitter](#) (uint32_t baseAddr)
Disables the LPUART transmitter.
- static bool [LPUART_HAL_IsTransmitterEnabled](#) (uint32_t baseAddr)
Gets the LPUART transmitter enabled/disabled configuration.
- static void [LPUART_HAL_EnableReceiver](#) (uint32_t baseAddr)
Enables the LPUART receiver.
- static void [LPUART_HAL_DisableReceiver](#) (uint32_t baseAddr)
Disables the LPUART receiver.
- static bool [LPUART_HAL_IsReceiverEnabled](#) (uint32_t baseAddr)
Gets the LPUART receiver enabled/disabled configuration.
- [lpuart_status_t LPUART_HAL_SetBaudRate](#) (uint32_t baseAddr, uint32_t sourceClockInHz,
uint32_t desiredBaudRate)
Configures the LPUART baud rate.
- static void [LPUART_HAL_SetBaudRateDivisor](#) (uint32_t baseAddr, uint32_t baudRateDivisor)
Sets the LPUART baud rate modulo divisor.
- void [LPUART_HAL_SetBitCountPerChar](#) (uint32_t baseAddr, [lpuart_bit_count_per_char_t](#) bit-
CountPerChar)
Configures the number of bits per character in the LPUART controller.
- void [LPUART_HAL_SetParityMode](#) (uint32_t baseAddr, [lpuart_parity_mode_t](#) parityModeType)
Configures parity mode in the LPUART controller.
- static void [LPUART_HAL_SetStopBitCount](#) (uint32_t baseAddr, [lpuart_stop_bit_count_t](#) stopBit-
Count)
Configures the number of stop bits in the LPUART controller.
- void [LPUART_HAL_SetTxRxInversionCmd](#) (uint32_t baseAddr, uint32_t rxInvert, uint32_t tx-
Invert)
Configures the transmit and receive inversion control in the LPUART controller.

LPUART Interrupts and DMA

- void [LPUART_HAL_SetIntMode](#) (uint32_t baseAddr, [lpuart_interrupt_t](#) interrupt, bool enable)
Configures the LPUART module interrupts to enable/disable various interrupt sources.
- bool [LPUART_HAL_GetIntMode](#) (uint32_t baseAddr, [lpuart_interrupt_t](#) interrupt)
Returns whether the LPUART module interrupts is enabled/disabled.
- static void [LPUART_HAL_SetTxDataRegEmptyIntCmd](#) (uint32_t baseAddr, bool enable)

LPUART HAL driver

Enable/Disable the transmission_complete_interrupt.

- static bool [LPUART_HAL_GetTxDataRegEmptyIntCmd](#) (uint32_t baseAddr)
Gets the configuration of the transmission_data_register_empty_interrupt enable setting.
- static void [LPUART_HAL_SetRxDataRegFullIntCmd](#) (uint32_t baseAddr, bool enable)
Enables the rx_data_register_full_interrupt.
- static bool [LPUART_HAL_GetRxDataRegFullIntCmd](#) (uint32_t baseAddr)
Gets the configuration of the rx_data_register_full_interrupt enable.

LPUART Transfer Functions

- static void [LPUART_HAL_Putchar](#) (uint32_t baseAddr, uint8_t data)
Sends the LPUART 8-bit character.
- void [LPUART_HAL_Putchar9](#) (uint32_t baseAddr, uint16_t data)
Sends the LPUART 9-bit character.
- [lpuart_status_t](#) [LPUART_HAL_Putchar10](#) (uint32_t baseAddr, uint16_t data)
Sends the LPUART 10-bit character (Note: Feature available on select LPUART instances).
- void [LPUART_HAL_Getchar](#) (uint32_t baseAddr, uint8_t *readData)
Gets the LPUART 8-bit character.
- void [LPUART_HAL_Getchar9](#) (uint32_t baseAddr, uint16_t *readData)
Gets the LPUART 9-bit character.
- [lpuart_status_t](#) [LPUART_HAL_Getchar10](#) (uint32_t baseAddr, uint16_t *readData)
Gets the LPUART 10-bit character.
- static void [LPUART_HAL_IdleConfig](#) (uint32_t baseAddr, [lpuart_idle_config_t](#) idleConfig)
Configures the number of idle characters that must be received before the IDLE flag is set.
- static [lpuart_idle_config_t](#) [LPUART_HAL_GetIdleconfig](#) (uint32_t baseAddr)
Gets the configuration of the number of idle characters that must be received before the IDLE flag is set.

LPUART Special Feature Configurations

- static void [LPUART_HAL_SetWaitModeOperation](#) (uint32_t baseAddr, [lpuart_operation_config_t](#) mode)
Configures the LPUART operation in wait mode (operates or stops operations in wait mode).
- [lpuart_operation_config_t](#) [LPUART_HAL_GetWaitModeOperationConfig](#) (uint32_t baseAddr)
Gets the LPUART operation in wait mode (operates or stops operations in wait mode).
- void [LPUART_HAL_SetLoopbackCmd](#) (uint32_t baseAddr, bool enable)
Configures the LPUART loopback operation (enable/disable loopback operation)
- void [LPUART_HAL_SetSingleWireCmd](#) (uint32_t baseAddr, bool enable)
Configures the LPUART single-wire operation (enable/disable single-wire mode)
- static void [LPUART_HAL_ConfigureTxdirectionInSinglewireMode](#) (uint32_t baseAddr, [lpuart_singlewire_txdir_t](#) direction)
Configures the LPUART transmit direction while in single-wire mode.
- [lpuart_status_t](#) [LPUART_HAL_PutReceiverInStandbyMode](#) (uint32_t baseAddr)
Places the LPUART receiver in standby mode.
- static void [LPUART_HAL_PutReceiverInNormalMode](#) (uint32_t baseAddr)
Places the LPUART receiver in a normal mode (disable standby mode operation).
- static bool [LPUART_HAL_IsReceiverInStandby](#) (uint32_t baseAddr)
Checks whether the LPUART receiver is in a standby mode.

- static void **LPUART_HAL_SelectReceiverWakeupMethod** (uint32_t baseAddr, **lpuart_wakeup_method_t** method)

LPUART receiver wakeup method (idle line or addr-mark) from standby mode.
- **lpuart_wakeup_method_t LPUART_HAL_GetReceiverWakeupMethod** (uint32_t baseAddr)

Gets the LPUART receiver wakeup method (idle line or addr-mark) from standby mode.
- void **LPUART_HAL_ConfigureIdleLineDetect** (uint32_t baseAddr, const **lpuart_idle_line_config_t** *config)

LPUART idle-line detect operation configuration (idle line bit-count start and wake up affect on IDLE status bit).
- static void **LPUART_HAL_SetBreakCharTransmitLength** (uint32_t baseAddr, **lpuart_break_char_length_t** length)

LPUART break character transmit length configuration In some LPUART instances, the user should disable the transmitter before calling this function.
- static void **LPUART_HAL_SetBreakCharDetectLength** (uint32_t baseAddr, **lpuart_break_char_length_t** length)

LPUART break character detect length configuration.
- static void **LPUART_HAL_QueueBreakCharToSend** (uint32_t baseAddr, bool enable)

LPUART transmit sends break character configuration.
- **lpuart_status_t LPUART_HAL_SetMatchAddressOperation** (uint32_t baseAddr, bool matchAddrMode1, bool matchAddrMode2, uint8_t matchAddrValue1, uint8_t matchAddrValue2, **lpuart_match_config_t** config)

LPUART configures match address mode control (Note: Feature available on select LPUART instances)
- static void **LPUART_HAL_ConfigureSendMsbFirstOperation** (uint32_t baseAddr, bool enable)

LPUART sends the MSB first configuration (Note: Feature available on select LPUART instances) In some LPUART instances, the user should disable the transmitter/receiver before calling this function.
- static void **LPUART_HAL_ConfigureReceiveResyncDisableOperation** (uint32_t baseAddr, bool enable)

LPUART disables re-sync of received data configuration (Note: Feature available on select LPUART instances).

LPUART Status Flags

- bool **LPUART_HAL_GetStatusFlag** (uint32_t baseAddr, **lpuart_status_flag_t** statusFlag)

LPUART get status flag.
- static bool **LPUART_HAL_IsTxDataRegEmpty** (uint32_t baseAddr)

Gets the LPUART Transmit data register empty flag.
- static bool **LPUART_HAL_IsRxDataRegFull** (uint32_t baseAddr)

Gets the LPUART receive data register full flag.
- static bool **LPUART_HAL_IsTxComplete** (uint32_t baseAddr)

Gets the LPUART transmission complete flag.
- **lpuart_status_t LPUART_HAL_ClearStatusFlag** (uint32_t baseAddr, **lpuart_status_flag_t** statusFlag)

LPUART clears an individual status flag (see lpuart_status_flag_t for list of status bits).
- void **LPUART_HAL_ClearAllNonAutoclearStatusFlags** (uint32_t baseAddr)

LPUART clears ALL status flags.

15.1.1 Data Structure Documentation

15.1.1.1 struct lpuart_idle_line_config_t

Data Fields

- unsigned `idleLineType`: 1
ILT, Idle bit count start: 0 - after start bit (default),
- unsigned `rxWakeIdleDetect`: 1
1 - after stop bit

15.1.1.0.36 Field Documentation

15.1.1.0.36.1 `unsigned lpuart_idle_line_config_t::rxWakeIdleDetect`

RWUID, Receiver Wake Up Idle Detect. IDLE status bit

15.1.2 Enumeration Type Documentation

15.1.2.1 enum lpuart_status_t

Enumerator

`kStatus_LPUART_BaudRateCalculationError` LPUART Baud Rate calculation error out of range.

`kStatus_LPUART_BaudRatePercentDiffExceeded` LPUART Baud Rate exceeds percentage difference.

`kStatus_LPUART_BitCountNotSupported` LPUART bit count configuration not supported.

`kStatus_LPUART_StopBitCountNotSupported` LPUART stop bit count configuration not supported.

`kStatus_LPUART_RxStandbyModeError` LPUART unable to place receiver in standby mode.

`kStatus_LPUART_ClearStatusFlagError` LPUART clear status flag error.

`kStatus_LPUART_MSBFirstNotSupported` LPUART MSB first feature not supported.

`kStatus_LPUART_Resync_NotSupported` LPUART resync disable operation not supported.

`kStatus_LPUART_TxNotDisabled` LPUART Transmitter not disabled before enabling feature.

`kStatus_LPUART_RxNotDisabled` LPUART Receiver not disabled before enabling feature.

`kStatus_LPUART_TxOrRxNotDisabled` LPUART Transmitter or Receiver not disabled.

`kStatus_LPUART_TxBusy` LPUART transmit still in progress.

`kStatus_LPUART_RxBusy` LPUART receive still in progress.

`kStatus_LPUART_NoTransmitInProgress` LPUART no transmit in progress.

`kStatus_LPUART_NoReceiveInProgress` LPUART no receive in progress.

`kStatus_LPUART_InvalidInstanceNumber` Invalid LPUART base address.

`kStatus_LPUART_InvalidBitSetting` Invalid setting for desired LPUART register bit field.

`kStatus_LPUART_OverSamplingNotSupported` LPUART oversampling not supported.

`kStatus_LPUART_BothEdgeNotSupported` LPUART both edge sampling not supported.

`kStatus_LPUART_Timeout` LPUART transfer timed out.

15.1.2.2 enum lpuart_stop_bit_count_t

Enumerator

kLpuartOneStopBit one stop bit

kLpuartTwoStopBit two stop bits

15.1.2.3 enum lpuart_parity_mode_t

Enumerator

kLpuartParityDisabled parity disabled

kLpuartParityEven parity enabled, type even, bit setting: PE|PT = 10

kLpuartParityOdd parity enabled, type odd, bit setting: PE|PT = 11

15.1.2.4 enum lpuart_bit_count_per_char_t

Enumerator

kLpuart8BitsPerChar 8-bit data characters

kLpuart9BitsPerChar 9-bit data characters

kLpuart10BitsPerChar 10-bit data characters

15.1.2.5 enum lpuart_operation_config_t

Enumerator

kLpuartOperates LPUART continues to operate normally.

kLpuartStops LPUART stops operation.

15.1.2.6 enum lpuart_wakeup_method_t

Enumerator

kLpuartIdleLineWake Idle-line wakes the LPUART receiver from standby.

kLpuartAddrMarkWake Addr-mark wakes LPUART receiver from standby.

15.1.2.7 enum lpuart_idle_line_select_t

Enumerator

kLpuartIdleLineAfterStartBit LPUART idle character bit count start after start bit.

kLpuartIdleLineAfterStopBit LPUART idle character bit count start after stop bit.

LPUART HAL driver

15.1.2.8 enum lpuart_break_char_length_t

The actual maximum bit times may vary depending on the LPUART instance.

Enumerator

kLpuartBreakChar10BitMinimum LPUART break char length 10 bit times (if M = 0, SBNS = 0) or 11 (if M = 1, SBNS = 0 or M = 0, SBNS = 1) or 12 (if M = 1, SBNS = 1 or M10 = 1, SNBS = 0) or 13 (if M10 = 1, SNBS = 1).

kLpuartBreakChar13BitMinimum LPUART break char length 13 bit times (if M = 0, SBNS = 0) or 14 (if M = 1, SBNS = 0 or M = 0, SBNS = 1) or 15 (if M = 1, SBNS = 1 or M10 = 1, SNBS = 0) or 16 (if M10 = 1, SNBS = 1)

15.1.2.9 enum lpuart_singlewire_txdir_t

Enumerator

kLpuartSinglewireTxdirIn LPUART Single Wire mode TXDIR input.

kLpuartSinglewireTxdirOut LPUART Single Wire mode TXDIR output.

15.1.2.10 enum lpuart_match_config_t

Enumerator

kLpuartAddressMatchWakeup LPUART Address Match Wakeup.

kLpuartIdleMatchWakeup LPUART Idle Match Wakeup.

kLpuartMatchOnAndMatchOff LPUART Match On and Match Off.

kLpuartEnablesRwuOnDataMatch LPUART Enables RWU on Data Match and Match On/Off for transmitter CTS input.

15.1.2.11 enum lpuart_ir_tx_pulsewidth_t

Enumerator

kLpuartIrThreeSixteenthsWidth 3/16 pulse

kLpuartIrOneSixteenthWidth 1/16 pulse

kLpuartIrOneThirtysecondsWidth 1/32 pulse

kLpuartIrOneFourthWidth 1/4 pulse

15.1.2.12 enum lpuart_idle_config_t

Enumerator

kLpuart_1_IdleChar 1 idle character

kLpuart_2_IdleChar 2 idle character
kLpuart_4_IdleChar 4 idle character
kLpuart_8_IdleChar 8 idle character
kLpuart_16_IdleChar 16 idle character
kLpuart_32_IdleChar 32 idle character
kLpuart_64_IdleChar 64 idle character
kLpuart_128_IdleChar 128 idle character

15.1.2.13 enum lpuart_cts_source_t

Configures the source of the CTS input.

Enumerator

kLpuartCtsSourcePin LPUART CTS input is the LPUART_CTS pin.
kLpuartCtsSourceInvertedReceiverMatch LPUART CTS input is the inverted Receiver Match result.

15.1.2.14 enum lpuart_cts_config_t

Enumerator

kLpuartCtsSampledOnEachCharacter LPUART CTS input is sampled at the start of each character.
kLpuartCtsSampledOnIdle LPUART CTS input is sampled when the transmitter is idle.

15.1.2.15 enum lpuart_status_flag_t

This provides constants for the LPUART status flags for use in the UART functions.

Enumerator

kLpuartTxDataRegEmpty Tx data register empty flag, sets when Tx buffer is empty.
kLpuartTxComplete Transmission complete flag, sets when transmission activity complete.
kLpuartRxDataRegFull Rx data register full flag, sets when the receive data buffer is full.
kLpuartIdleLineDetect Idle line detect flag, sets when idle line detected.
kLpuartRxOverrun Rrx Overrun, sets when new data is received before data is read from receive register.
kLpuartNoiseDetect Rrx takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kLpuartFrameErr Frame error flag, sets if logic 0 was detected where stop bit expected.
kLpuartParityErr If parity enabled, sets upon parity error detection.

LPUART HAL driver

kLpuartLineBreakDetect LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.

kLpuartRxActiveEdgeDetect Rx pin active edge interrupt flag, sets when active edge detected.

kLpuartRxActive Receiver Active Flag (RAF), sets at beginning of valid start bit.

15.1.2.16 enum lpuart_interrupt_t

Enumerator

kLpuartIntLinBreakDetect LIN break detect.

kLpuartIntRxActiveEdge RX Active Edge.

kLpuartIntTxDataRegEmpty Transmit data register empty.

kLpuartIntTxComplete Transmission complete.

kLpuartIntRxDataRegFull Receiver data register full.

kLpuartIntIdleLine Idle line.

kLpuartIntRxOverrun Receiver Overrun.

kLpuartIntNoiseErrFlag Noise error flag.

kLpuartIntFrameErrFlag Framing error flag.

kLpuartIntParityErrFlag Parity error flag.

15.1.3 Function Documentation

15.1.3.1 void LPUART_HAL_Init (uint32_t baseAddr)

Parameters

<i>baseAddr</i>	LPUART base address.
-----------------	----------------------

15.1.3.2 void LPUART_HAL_EnableTransmitter (uint32_t baseAddr)

Parameters

<i>baseAddr</i>	LPUART base address.
-----------------	----------------------

15.1.3.3 static void LPUART_HAL_DisableTransmitter (uint32_t baseAddr) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

15.1.3.4 static bool LPUART_HAL_IsTransmitterEnabled (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

State of LPUART transmitter enable(1)/disable(0)

15.1.3.5 static void LPUART_HAL_EnableReceiver (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

15.1.3.6 static void LPUART_HAL_DisableReceiver (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

15.1.3.7 static bool LPUART_HAL_IsReceiverEnabled (*uint32_t baseAddr*) [inline], [static]

Parameters

LPUART HAL driver

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

State of LPUART receiver enable(1)/disable(0)

15.1.3.8 lpuart_status_t LPUART_HAL_SetBaudRate (uint32_t *baseAddr*, uint32_t *sourceClockInHz*, uint32_t *desiredBaudRate*)

In some LPUART instances the user must disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address.
<i>sourceClockInHz</i>	LPUART source input clock in Hz.
<i>desiredBaudRate</i>	LPUART desired baud rate.

Returns

An error code or kStatus_Success

15.1.3.9 static void LPUART_HAL_SetBaudRateDivisor (uint32_t *baseAddr*, uint32_t *baudRateDivisor*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address.
<i>baudRateDivisor</i>	The baud rate modulo division "SBR"

15.1.3.10 void LPUART_HAL_SetBitCountPerChar (uint32_t *baseAddr*, lpuart_bit_count_per_char_t *bitCountPerChar*)

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address.
<i>bitCountPerChar</i>	Number of bits per char (8, 9, or 10, depending on the LPUART instance)

15.1.3.11 void LPUART_HAL_SetParityMode (*uint32_t baseAddr, lpuart_parity_mode_t parityModeType*)

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address.
<i>parityModeType</i>	Parity mode (enabled, disable, odd, even - see parity_mode_t struct)

15.1.3.12 static void LPUART_HAL_SetStopBitCount (*uint32_t baseAddr, lpuart_stop_bit_count_t stopBitCount*) [inline], [static]

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address.
<i>stopBitCount</i>	Number of stop bits (1 or 2 - see lpuart_stop_bit_count_t struct)

Returns

An error code (an unsupported setting in some LPUARTs) or kStatus_Success

15.1.3.13 void LPUART_HAL_SetTxRxInversionCmd (*uint32_t baseAddr, uint32_t rxInvert, uint32_t txInvert*)

This function should only be called when the LPUART is between transmit and receive packets.

LPUART HAL driver

Parameters

<i>baseAddr</i>	LPUART base address.
<i>rxInvert</i>	Enable (1) or disable (0) receive inversion
<i>txInvert</i>	Enable (1) or disable (0) transmit inversion

15.1.3.14 void LPUART_HAL_SetIntMode (uint32_t *baseAddr*, lpuart_interrupt_t *interrupt*, bool *enable*)

Parameters

<i>baseAddr</i>	LPUART module base address.
<i>interrupt</i>	LPUART interrupt configuration data.
<i>enable</i>	true: enable, false: disable.

15.1.3.15 bool LPUART_HAL_GetIntMode (uint32_t *baseAddr*, lpuart_interrupt_t *interrupt*)

Parameters

<i>baseAddr</i>	LPUART module base address.
<i>interrupt</i>	LPUART interrupt configuration data.

Returns

true: enable, false: disable.

15.1.3.16 static void LPUART_HAL_SetTxDataRegEmptyIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	true: enable, false: disable.

15.1.3.17 static bool LPUART_HAL_GetTxDataRegEmptyIntCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

Bit setting of the interrupt enable bit

15.1.3.18 static void LPUART_HAL_SetRxDataRegFullIntCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	true: enable, false: disable.

15.1.3.19 static bool LPUART_HAL_GetRxDataRegFullIntCmd (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

Bit setting of the interrupt enable bit

15.1.3.20 static void LPUART_HAL_Putchar (*uint32_t baseAddr, uint8_t data*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART Instance
<i>data</i>	data to send (8-bit)

15.1.3.21 void LPUART_HAL_Putchar9 (*uint32_t baseAddr, uint16_t data*)

LPUART HAL driver

Parameters

<i>baseAddr</i>	LPUART Instance
<i>data</i>	data to send (9-bit)

15.1.3.22 **lpuart_status_t LPUART_HAL_Putchar10 (uint32_t *baseAddr*, uint16_t *data*)**

Parameters

<i>baseAddr</i>	LPUART Instance
<i>data</i>	data to send (10-bit)

Returns

An error code or kStatus_Success

15.1.3.23 **void LPUART_HAL_Getchar (uint32_t *baseAddr*, uint8_t * *readData*)**

Parameters

<i>baseAddr</i>	LPUART base address
<i>readData</i>	data read from receive (8-bit)

15.1.3.24 **void LPUART_HAL_Getchar9 (uint32_t *baseAddr*, uint16_t * *readData*)**

Parameters

<i>baseAddr</i>	LPUART base address
<i>readData</i>	data read from receive (9-bit)

15.1.3.25 **lpuart_status_t LPUART_HAL_Getchar10 (uint32_t *baseAddr*, uint16_t * *readData*)**

Parameters

<i>baseAddr</i>	LPUART base address
<i>readData</i>	data read from receive (10-bit)

Returns

An error code or kStatus_Success

15.1.3.26 static void LPUART_HAL_IdleConfig (uint32_t *baseAddr*, lpuart_idle_config_t *idleConfig*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>idle_config</i>	idle characters configuration

15.1.3.27 static lpuart_idle_config_t LPUART_HAL_GetIdleconfig (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

idle characters configuration

15.1.3.28 static void LPUART_HAL_SetWaitModeOperation (uint32_t *baseAddr*, lpuart_operation_config_t *mode*) [inline], [static]

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address
<i>mode</i>	LPUART wait mode operation - operates or stops to operate in wait mode.

LPUART HAL driver

15.1.3.29 lpuart_operation_config_t LPUART_HAL_GetWaitModeOperationConfig (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

LPUART wait mode operation configuration - kLpuartOperates or KLpuartStops in wait mode

15.1.3.30 void LPUART_HAL_SetLoopbackCmd (uint32_t *baseAddr*, bool *enable*)

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	LPUART loopback mode - disabled (0) or enabled (1)

15.1.3.31 void LPUART_HAL_SetSingleWireCmd (uint32_t *baseAddr*, bool *enable*)

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	LPUART loopback mode - disabled (0) or enabled (1)

15.1.3.32 static void LPUART_HAL_ConfigureTxdirInSinglewireMode (uint32_t *baseAddr*, lpuart_singlewire_txdir_t *direction*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>direction</i>	LPUART single-wire transmit direction - input or output

LPUART HAL driver

15.1.3.33 lpuart_status_t LPUART_HAL_PutReceiverInStandbyMode (uint32_t *baseAddr*)

In some LPUART instances, before placing LPUART in standby mode, first determine whether the receiver is set to wake on idle or whether it is already in idle state. NOTE that the RWU should only be set with C1[WAKE] = 0 (wakeup on idle) if the channel is currently not idle. This can be determined by the S2[RAF] flag. If it is set to wake up an IDLE event and the channel is already idle, it is possible that the LPUART will discard data since data must be received (or a LIN break detect) after an IDLE is detected and before IDLE is allowed to reasserted.

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

Error code or kStatus_Success

15.1.3.34 static void LPUART_HAL_PutReceiverInNormalMode (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

15.1.3.35 static bool LPUART_HAL_IsReceiverInStandby (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

LPUART in normal mode (0) or standby (1)

15.1.3.36 static void LPUART_HAL_SelectReceiverWakeupMethod (uint32_t *baseAddr*, lpuart_wakeup_method_t *method*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>method</i>	LPUART wakeup method: 0 - Idle-line wake (default), 1 - addr-mark wake

15.1.3.37 **lpuart_wakeup_method_t LPUART_HAL_GetReceiverWakeupMethod (uint32_t *baseAddr*)**

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

LPUART wakeup method: kLpuartIdleLineWake: 0 - Idle-line wake (default), kLpuartAddrMarkWake: 1 - addr-mark wake

15.1.3.38 **void LPUART_HAL_ConfigureIdleLineDetect (uint32_t *baseAddr*, const lpuart_idle_line_config_t * *config*)**

In some LPUART instances, the user should disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address
<i>config</i>	LPUART configuration data for idle line detect operation

15.1.3.39 **static void LPUART_HAL_SetBreakCharTransmitLength (uint32_t *baseAddr*, lpuart_break_char_length_t *length*) [inline], [static]**

Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address
<i>length</i>	LPUART break character length setting: 0 - minimum 10-bit times (default), 1 - minimum 13-bit times

LPUART HAL driver

**15.1.3.40 static void LPUART_HAL_SetBreakCharDetectLength (uint32_t *baseAddr*,
 lpuart_break_char_length_t *length*) [inline], [static]**

Parameters

<i>baseAddr</i>	LPUART base address
<i>length</i>	LPUART break character length setting: 0 - minimum 10-bit times (default), 1 - minimum 13-bit times

15.1.3.41 static void LPUART_HAL_QueueBreakCharToSend (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	LPUART normal/queue break char - disabled (normal mode, default: 0) or enabled (queue break char: 1)

15.1.3.42 lpuart_status_t LPUART_HAL_SetMatchAddressOperation (uint32_t *baseAddr*, bool *matchAddrMode1*, bool *matchAddrMode2*, uint8_t *matchAddrValue1*, uint8_t *matchAddrValue2*, lpuart_match_config_t *config*)

Parameters

<i>baseAddr</i>	LPUART base address
<i>matchAddr-Mode1</i>	MAEN1: match address mode1 enable (1)/disable (0)
<i>matchAddr-Mode2</i>	MAEN2: match address mode2 enable (1)/disable (0)
<i>matchAddr-Value1</i>	MA: match address value to program into match address register 1
<i>matchAddr-Value2</i>	MA: match address value to program into match address register 2
<i>config</i>	MATCFG: Configures the match addressing mode used.

LPUART HAL driver

Returns

An error code or kStatus_Success

15.1.3.43 static void LPUART_HAL_ConfigureSendMsbFirstOperation (uint32_t baseAddr, bool enable) [inline], [static]

Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	MSB first mode configuration, MSBF: 0 - LSB (default, feature disabled), 1 - MSB (feature enabled)

15.1.3.44 static void LPUART_HAL_ConfigureReceiveResyncDisableOperation (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
<i>enable</i>	disable re-sync of received data word configuration, RESYNCDIS: 0 - re-sync of received data word (default, feature disabled), 1 - disable the re-sync (feature enabled)

15.1.3.45 bool LPUART_HAL_GetStatusFlag (uint32_t *baseAddr*, Ipuart_status_flag_t *statusFlag*)

Parameters

<i>baseAddr</i>	LPUART base address
<i>statusFlag</i>	The status flag to query

15.1.3.46 static bool LPUART_HAL_IsTxDataRegEmpty (uint32_t *baseAddr*) [inline], [static]

This function returns the state of the LPUART Transmit data register empty flag.

Parameters

<i>baseAddr</i>	LPUART module base address.
-----------------	-----------------------------

Returns

The status of Transmit data register empty flag, which is set when transmit buffer is empty.

15.1.3.47 static bool LPUART_HAL_IsRxDataRegFull (uint32_t *baseAddr*) [inline], [static]

LPUART HAL driver

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

Status of the receive data register full flag, sets when the receive data buffer is full.

15.1.3.48 static bool LPUART_HAL_IsTxComplete (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

Returns

Status of Transmission complete flag, sets when transmitter is idle (transmission activity complete)

15.1.3.49 lpuart_status_t LPUART_HAL_ClearStatusFlag (uint32_t *baseAddr*, lpuart_status_flag_t *statusFlag*)

Parameters

<i>baseAddr</i>	LPUART base address
<i>statusFlag</i>	Desired LPUART status flag to clear

Returns

An error code or kStatus_Success

15.1.3.50 void LPUART_HAL_ClearAllNonAutoclearStatusFlags (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	LPUART base address
-----------------	---------------------

15.2 LPUART peripheral Driver

This chapter describes the programming interface of the LPUART Peripheral driver.

Data Structures

- struct `lpuart_state_t`
Runtime state of the LPUART driver. [More...](#)
- struct `lpuart_user_config_t`
LPUART configuration structure. [More...](#)

TypeDefs

- typedef `lpuart_status_t(* lpuart_rx_callback_t)`(`uint8_t *rxByte, void *param`)
LPUART receive callback function type.

LPUART Driver

- `lpuart_status_t LPUART_DRV_Init` (`uint32_t instance, lpuart_state_t *lpuartStatePtr, const lpuart_user_config_t *lpuartUserConfig`)
Initializes a LPUART operation instance.
- `void LPUART_DRV_Deinit` (`uint32_t instance`)
Shuts down the LPUART by disabling interrupts and transmitter/receiver.
- `lpuart_status_t LPUART_DRV_SendDataBlocking` (`uint32_t instance, const uint8_t *txBuff, uint32_t txSize, uint32_t timeout`)
Sends data out through the LPUART module using a blocking method.
- `lpuart_status_t LPUART_DRV_SendData` (`uint32_t instance, const uint8_t *txBuff, uint32_t txSize`)
Sends data out through the LPUART module using a non-blocking method.
- `lpuart_status_t LPUART_DRV_GetTransmitStatus` (`uint32_t instance, uint32_t *bytesRemaining`)
Returns whether the previous transmit is complete.
- `lpuart_status_t LPUART_DRV_AbortSendingData` (`uint32_t instance`)
Terminates an asynchronous transmission early.
- `lpuart_status_t LPUART_DRV_ReceiveDataBlocking` (`uint32_t instance, uint8_t *rxBuff, uint32_t rxSize, uint32_t timeout`)
Gets data from the LPUART module by using a blocking method.
- `lpuart_status_t LPUART_DRV_ReceiveData` (`uint32_t instance, uint8_t *rxBuff, uint32_t rxSize`)
Gets data from the LPUART module by using a non-blocking method.
- `lpuart_status_t LPUART_DRV_GetReceiveStatus` (`uint32_t instance, uint32_t *bytesRemaining`)
Returns whether the previous receive is complete.
- `lpuart_status_t LPUART_DRV_AbortReceivingData` (`uint32_t instance`)
Terminates an asynchronous receive early.

LPUART peripheral Driver

15.2.0.51 LPUART Peripheral Driver

Overview

The LPUART peripheral driver transfers data to and from external devices on the Low Power Universal Asynchronous Receiver/Transmitter (LPUART) serial bus. It provides a way to transmit or receive buffers of data with calls to a single function.

Device structures

The driver uses instantiations of the lpuart_tx_state_t and the lpuart_rx_state_t structure to maintain the current state of a particular LPUART instance module driver. The caller provides memory for the driver state structures during the initialization as the driver itself does not statically allocate memory. The structures are provided below:

```
// Runtime transmit state of the LPUART driver.
typedef struct LpuartTxState {
    uint32_t instance;
    bool isTransmitInProgress;
    bool isTransmitAsync;
    const uint8_t * sendBuffer;
    size_t remainingSendByteCount;
    size_t transmittedByteCount;
    sync_object_t irqSync;
    uint8_t fifoEntryCount;
} lpuart_tx_state_t;

// Runtime receive state of the LPUART driver.
typedef struct LpuartRxState {
    uint32_t instance;
    bool isReceiveInProgress;
    bool isReceiveAsync;
    uint8_t * receiveBuffer;
    size_t remainingReceiveByteCount;
    size_t receivedByteCount;
    sync_object_t irqSync;
    uint8_t fifoEntryCount;
} lpuart_rx_state_t;
```

Initialization

To initialize the LPUART driver, call the [LPUART_DRV_Init\(\)](#) function and pass the instance number of the LPUART peripheral you want to use. For instance, to use LPUART0 pass a value of 0 to the initialization function. In addition, you also have to pass a user configuration structure [lpuart_user_config_t](#) shown here:

```
// LPUART configuration structure for user
typedef struct LpuartUserConfig {
    uint32_t baudRate;
    lpuart_parity_mode_t parityMode;
    lpuart_stop_bit_count_t stopBitCount;
    lpuart_bit_count_per_char_t bitCountPerChar;
} lpuart_user_config_t;
```

Typically the user configures the `lpuart_user_config_t` instantiation as an 8-bit-char, no-parity, 1-stop-bit (8-n-1) with a baud rate of 9600 bps. The user can easily modify the `lpuart_user_config_t` instantiation to configure the LPUART peripheral to a different baud rate or character transfer features. This is a code example to set up a user LPUART configuration instantiation:

```
lpuart_user_config_t lpuartConfig;
lpuartConfig.baudRate = 9600;
lpuartConfig.bitCountPerChar = kLpuart8BitsPerChar;
lpuartConfig.parityMode = kLpuartParityDisabled;
lpuartConfig.stopBitCount = kLpuartOneStopBit;
```

Transfers

The driver implements transmit and receive functions to transfer buffers of data. The driver supports two different modes for transferring data: blocking and non-blocking.

The blocking transmit and receive functions are the `LPUART_DRV_SendData()` and the `LPUART_DRV_V_ReceiveData()`.

The non-blocking (async) transmit and receive functions are the `LPUART_DRV_SendData_async()` and the `LPUART_DRV_ReceiveData_async()`.

In all of these cases, the functions are interrupt driven.

15.2.1 Data Structure Documentation

15.2.1.1 struct `lpuart_state_t`

Note that the caller provides memory for the driver state structures during initialization because the driver does not statically allocate memory.

Data Fields

- const uint8_t * `txBuff`
The buffer of data being sent.
- uint8_t * `rxBuff`
The buffer of received data.
- volatile size_t `txSize`
The remaining number of bytes to be transmitted.
- volatile size_t `rxSize`
The remaining number of bytes to be received.
- volatile bool `isTxBusy`
True if there is an active transmit.
- volatile bool `isRxBusy`
True if there is an active receive.
- volatile bool `isTxBlocking`
True if transmit is blocking transaction.
- volatile bool `isRxBlocking`

LPUART peripheral Driver

True if receive is blocking transaction.

- **semaphore_t txIrqSync**
Used to wait for ISR to complete its Tx business.
- **semaphore_t rxIrqSync**
Used to wait for ISR to complete its Rx business.
- **lpuart_rx_callback_t rxCallback**
Callback to invoke after receiving byte.
- **void * rxCallbackParam**
Receive callback parameter pointer.

15.2.1.1.0.37 Field Documentation

15.2.1.1.0.37.1 **const uint8_t* lpuart_state_t::txBuff**

15.2.1.1.0.37.2 **uint8_t* lpuart_state_t::rxBuff**

15.2.1.1.0.37.3 **volatile size_t lpuart_state_t::txSize**

15.2.1.1.0.37.4 **volatile size_t lpuart_state_t::rxSize**

15.2.1.1.0.37.5 **volatile bool lpuart_state_t::isTxBusy**

15.2.1.1.0.37.6 **volatile bool lpuart_state_t::isRxBusy**

15.2.1.1.0.37.7 **volatile bool lpuart_state_t::isTxBlocking**

15.2.1.1.0.37.8 **volatile bool lpuart_state_t::isRxBlocking**

15.2.1.1.0.37.9 **semaphore_t lpuart_state_t::txIrqSync**

15.2.1.1.0.37.10 **semaphore_t lpuart_state_t::rxIrqSync**

15.2.1.1.0.37.11 **lpuart_rx_callback_t lpuart_state_t::rxCallback**

15.2.1.1.0.37.12 **void* lpuart_state_t::rxCallbackParam**

15.2.1.2 struct lpuart_user_config_t

Data Fields

- **uint32_t baudRate**
LPUART baud rate.
- **lpuart_parity_mode_t parityMode**
parity mode, disabled (default), even, odd
- **lpuart_stop_bit_count_t stopBitCount**
number of stop bits, 1 stop bit (default) or 2 stop bits
- **lpuart_bit_count_per_char_t bitCountPerChar**
number of bits, 8-bit (default) or 9-bit in a char (up to 10-bits in some LPUART instances).

15.2.1.2.0.38 Field Documentation

15.2.1.2.0.38.1 `lpuart_bit_count_per_char_t lpuart_user_config_t::bitCountPerChar`

15.2.2 Typedef Documentation

15.2.2.1 `typedef lpuart_status_t(* lpuart_rx_callback_t)(uint8_t *rxByte, void *param)`

15.2.3 Function Documentation

15.2.3.1 `lpuart_status_t LPUART_DRV_Init (uint32_t instance, lpuart_state_t * lpuartStatePtr, const lpuart_user_config_t * lpuartUserConfig)`

The caller provides memory for the driver state structures during initialization. The user must select the LPUART clock source in the application to initialize the LPUART.

LPUART peripheral Driver

Parameters

<i>instance</i>	LPUART instance number
<i>lpuartUserConfig</i>	user configuration structure of type lpuart_user_config_t
<i>lpuartStatePtr</i>	pointer to the LPUART driver state structure

Returns

An error code or kStatus_LPUART_Success

15.2.3.2 void LPUART_DRV_Deinit(uint32_t *instance*)

Parameters

<i>instance</i>	LPUART instance number
-----------------	------------------------

15.2.3.3 lpuart_status_t LPUART_DRV_SendDataBlocking(uint32_t *instance*, const uint8_t * *txBuff*, uint32_t *txSize*, uint32_t *timeout*)

Blocking means that the function does not return until the transmission is complete.

Parameters

<i>instance</i>	LPUART instance number
<i>txBuff</i>	source buffer containing 8-bit data chars to send
<i>txSize</i>	the number of bytes to send
<i>timeout</i>	timeout value for RTOS abstraction sync control

Returns

An error code or kStatus_LPUART_Success

15.2.3.4 lpuart_status_t LPUART_DRV_SendData(uint32_t *instance*, const uint8_t * *txBuff*, uint32_t *txSize*)

This enables an async method for transmitting data. When used with a non-blocking receive, the LPUART can perform a full duplex operation. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete.

Parameters

<i>instance</i>	LPUART instance number
<i>txBuff</i>	source buffer containing 8-bit data chars to send
<i>txSize</i>	the number of bytes to send

Returns

An error code or kStatus_LPUART_Success

15.2.3.5 lpuart_status_t LPUART_DRV_GetTransmitStatus (uint32_t *instance*, uint32_t * *bytesRemaining*)

Parameters

<i>instance</i>	LPUART instance number
<i>bytesRemaining</i>	Pointer to value that will be filled in with the number of bytes that have been sent in the active transfer

Return values

<i>kStatus_LPUART_Success</i>	The transmit has completed successfully.
<i>kStatus_LPUART_TxBusy</i>	The transmit is still in progress. <i>bytesTransmitted</i> will be filled with the number of bytes that have been transmitted so far.

15.2.3.6 lpuart_status_t LPUART_DRV_AbortSendingData (uint32_t *instance*)

Parameters

<i>instance</i>	LPUART instance number
-----------------	------------------------

Return values

<i>kStatus_LPUART_Success</i>	The transmit was successful.
-------------------------------	------------------------------

LPUART peripheral Driver

15.2.3.7 lpuart_status_t LPUART_DRV_ReceiveDataBlocking (uint32_t *instance*, uint8_t * *rxBuff*, uint32_t *rxSize*, uint32_t *timeout*)

Blocking means that the function does not return until the receive is complete.

Parameters

<i>instance</i>	LPUART instance number
<i>rxBuff</i>	buffer containing 8-bit read data chars received
<i>rxSize</i>	the number of bytes to receive
<i>timeout</i>	timeout value for RTOS abstraction sync control

Returns

An error code or kStatus_LPUART_Success

15.2.3.8 lpuart_status_t LPUART_DRV_ReceiveData (uint32_t *instance*, uint8_t * *rxBuff*, uint32_t *rxSize*)

This enables an async method for receiving data. When used with a non-blocking transmission, the LPUART can perform a full duplex operation. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the receive is complete.

Parameters

<i>instance</i>	LPUART instance number
<i>rxBuff</i>	buffer containing 8-bit read data chars received
<i>rxSize</i>	the number of bytes to receive

Returns

An error code or kStatus_LPUART_Success

15.2.3.9 lpuart_status_t LPUART_DRV_GetReceiveStatus (uint32_t *instance*, uint32_t * *bytesRemaining*)

Parameters

<i>instance</i>	LPUART instance number
-----------------	------------------------

LPUART peripheral Driver

<i>bytesRemaining</i>	pointer to value that is filled with the number of bytes that still need to be received in the active transfer.
-----------------------	---

Return values

<i>kStatus_LPUART_Success</i>	the receive has completed successfully.
<i>kStatus_LPUART_RxBusy</i>	the receive is still in progress. <i>bytesReceived</i> will be filled with the number of bytes that have been received so far.

15.2.3.10 **lpuart_status_t LPUART_DRV_AbortReceivingData (uint32_t instance)**

Parameters

<i>instance</i>	LPUART instance number
-----------------	------------------------

Return values

<i>kStatus_LPUART_Success</i>	The receive was successful.
-------------------------------	-----------------------------

15.3 LPUART Types Definitions

This chapter describes the LPUART type definitions.

Chapter 16

Memory Protection Unit (MPU)

The Kinetis SDK provides both HAL and Peripheral drivers for the Memory Protection Unit (MPU) block of Kinetis devices.

Modules

- [MPU HAL driver](#)

This part describes the programming interface of the MPU HAL driver.

- [MPU Peripheral Driver](#)

This part describes the programming interface of the MPU Peripheral driver.

16.1 MPU HAL driver

This chapter describes the programming interface of the MPU HAL driver.

Enumerations

- enum `mpu_region_num` {
 `kMPURegionNum00` = 0U,
 `kMPURegionNum01` = 1U,
 `kMPURegionNum02` = 2U,
 `kMPURegionNum03` = 3U,
 `kMPURegionNum04` = 4U,
 `kMPURegionNum05` = 5U,
 `kMPURegionNum06` = 6U,
 `kMPURegionNum07` = 7U,
 `kMPURegionNum08` = 8U,
 `kMPURegionNum09` = 9U,
 `kMPURegionNum10` = 10U,
 `kMPURegionNum11` = 11U }
 MPU region number region0~region11.
- enum `mpu_error_addr_reg` {
 `kMPUErrorAddrReg00` = 0U,
 `kMPUErrorAddrReg01` = 1U,
 `kMPUErrorAddrReg02` = 2U,
 `kMPUErrorAddrReg03` = 3U,
 `kMPUErrorAddrReg04` = 4U }
 MPU error address register0~4.
- enum `mpu_error_detail_reg` {
 `kMPUErrorDetailReg00` = 0U,
 `kMPUErrorDetailReg01` = 1U,
 `kMPUErrorDetailReg02` = 2U,
 `kMPUErrorDetailReg03` = 3U,
 `kMPUErrorDetailReg04` = 4U }
 MPU error detail register0~4.
- enum `mpu_error_access_type` {
 `kMPUReadErrorType` = 0U,
 `kMPUWriteErrorType` = 1U }
 MPU access error.
- enum `mpu_error_attributes` {
 `kMPUUserModeInstructionAccess` = 0U,
 `kMPUUserModeDataAccess` = 1U,
 `kMPUSupervisorModeInstructionAccess` = 2U,
 `kMPUSupervisorModeDataAccess` = 3U }
 MPU access error attributes.
- enum `mpu_access_mode` {
 `kMPUAccessInUserMode` = 0U,

- ```
kMPUAccessInSupervisorMode = 1U }
```

*access MPU in which mode.*
- enum `mpu_master_num` {
`kMPUMaster00` = 0U,  
`kMPUMaster01` = 1U,  
`kMPUMaster02` = 2U,  
`kMPUMaster03` = 3U,  
`kMPUMaster04` = 4U,  
`kMPUMaster05` = 5U,  
`kMPUMaster06` = 6U,  
`kMPUMaster07` = 7U }

*MPU master number.*
- enum `mpu_error_access_control` {
`kMPUNoRegionHitError` = 0U,  
`kMPUNoneOverlappRegionError` = 1U,  
`kMPUOverlappRegionError` = 2U }

*MPU error access control detail.*
- enum `mpu_supervisor_access_rights` {
`kMPUSupervisorReadWriteExecute` = 0U,  
`kMPUSupervisorReadExecute` = 1U,  
`kMPUSupervisorReadWrite` = 2U,  
`kMPUSupervisorEqualToUsermode` = 3U }

*MPU access rights in supervisor mode for master0~master3.*
- enum `mpu_user_access_rights` {
`kMPUUserNoAccessRights` = 0U,  
`kMPUUserExecute` = 1U,  
`kMPUUserWrite` = 2U,  
`kMPUUserWriteExecute` = 3U,  
`kMPUUserRead` = 4U,  
`kMPUUserReadExecute` = 5U,  
`kMPUUserReadWrite` = 6U,  
`kMPUUserReadWriteExecute` = 7U }

*MPU access rights in user mode for master0~master3.*
- enum `mpu_process_identifier_value` {
`kMPUIIdentifierDisable` = 0U,  
`kMPUIIdentifierEnable` = 1U }

*MPU process identifier.*
- enum `mpu_access_control` {
`kMPUAccessDisable` = 0U,  
`kMPUAccessEnable` = 1U }

*MPU access control for master4~master7.*
- enum `mpu_access_type` {
`kMPUAccessRead` = 0U,  
`kMPUAccessWrite` = 1U }

*MPU access type for master4~master7.*
- enum `mpu_region_valid` {
`kMPURegionInvalid` = 0U,

## MPU HAL driver

```
kMPURegionValid = 1U }
 MPU access region valid.
• enum mpu_status_t {
 kStatus_MPU_Success = 0x0U,
 kStatus_MPU_NotInitialized = 0x1U,
 kStatus_MPU_NullArgument = 0x2U }
 MPU status return codes.
```

## MPU HAL.

- static void **MPU\_HAL\_Enable** (uint32\_t baseAddr)  
*Enables the MPU module operation.*
- static void **MPU\_HAL\_Disable** (uint32\_t baseAddr)  
*Disables the MPU module operation.*
- static bool **MPU\_HAL\_IsEnabled** (uint32\_t baseAddr)  
*Checks whether the MPU module is enabled.*
- static uint32\_t **MPU\_HAL\_GetNumberOfRegions** (uint32\_t baseAddr)  
*Returns the total region number.*
- static uint32\_t **MPU\_HAL\_GetNumberOfSlaves** (uint32\_t baseAddr)  
*Returns MPU slave sports.*
- static uint32\_t **MPU\_HAL\_GetHardwareRevisionLevel** (uint32\_t baseAddr)  
*Returns hardware level info.*
- static uint32\_t **MPU\_HAL\_GetErrorAccessAddr** (uint32\_t baseAddr, **mpu\_error\_addr\_reg** regNum)  
*Returns hardware level info.*
- static uint8\_t **MPU\_HAL\_GetErrorSlaveSports** (uint32\_t baseAddr)  
*Returns error access slaves sports.*
- static **mpu\_error\_access\_type** **MPU\_HAL\_GetErrorAccessType** (uint32\_t baseAddr, **mpu\_error\_detail\_reg** errorDetailRegNum)  
*Returns error access address.*
- static **mpu\_error\_attributes** **MPU\_HAL\_GetErrorAttributes** (uint32\_t baseAddr, **mpu\_error\_detail\_reg** errorDetailRegNum)  
*Returns error access attributes.*
- static **mpu\_master\_num** **MPU\_HAL\_GetErrorMasterNum** (uint32\_t baseAddr, **mpu\_error\_detail\_reg** errorDetailRegNum)  
*Returns error access master number.*
- static uint32\_t **MPU\_HAL\_GetErrorProcessIdentifier** (uint32\_t baseAddr, **mpu\_error\_detail\_reg** errorDetailRegNum)  
*Returns error process identifier.*
- static **mpu\_error\_access\_control** **MPU\_HAL\_GetErrorAccessControl** (uint32\_t baseAddr, **mpu\_error\_detail\_reg** errorDetailRegNum)  
*Returns error access control.*
- static uint32\_t **MPU\_HAL\_GetRegionStartAddr** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Returns the region start address.*
- static void **MPU\_HAL\_SetRegionStartAddr** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, uint32\_t startAddr)  
*Sets region start address.*

- static uint32\_t **MPU\_HAL\_GetRegionEndAddr** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Returns region end address.*
- static void **MPU\_HAL\_SetRegionEndAddr** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, uint32\_t endAddr)  
*Sets region end address.*
- static uint32\_t **MPU\_HAL\_GetAllMastersAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Returns all masters access permission for a specific region.*
- static void **MPU\_HAL\_SetAllMastersAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, uint32\_t accessRights)  
*Sets all masters access permission for a specific region.*
- static **mpu\_supervisor\_access\_rights** **MPU\_HAL\_GetM0SupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Gets M0 access permission in supervisor mode.*
- static **mpu\_user\_access\_rights** **MPU\_HAL\_GetM0UserAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Gets M0 access permission in user mode.*
- static void **MPU\_HAL\_SetM0SupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_supervisor\_access\_rights** accessRights)  
*Sets M0 access permission in supervisor mode.*
- static void **MPU\_HAL\_SetM0UserAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_user\_access\_rights** accessRights)  
*Sets M0 access permission in user mode.*
- static bool **MPU\_HAL\_IsM0ProcessIdentifierEnabled** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Checks whether the M0 process identifier is enabled in region hit evaluation.*
- static void **MPU\_HAL\_SetM0ProcessIdentifierValue** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_process\_identifier\_value** identifierValue)  
*Sets the M0 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*
- static **mpu\_supervisor\_access\_rights** **MPU\_HAL\_GetM1SupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Gets M1 access permission in supervisor mode.*
- static **mpu\_user\_access\_rights** **MPU\_HAL\_GetM1UserAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Gets M1 access permission in user mode.*
- static void **MPU\_HAL\_SetM1SupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_supervisor\_access\_rights** accessRights)  
*Sets M1 access permission in supervisor mode.*
- static void **MPU\_HAL\_SetM1UserAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_user\_access\_rights** accessRights)  
*Sets M1 access permission in user mode.*
- static bool **MPU\_HAL\_IsM1ProcessIdentifierEnabled** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Checks whether M1 process identifier enabled in region hit evaluation.*
- static void **MPU\_HAL\_SetM1ProcessIdentifierValue** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_process\_identifier\_value** identifierValue)  
*Sets the M1 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*
- static **mpu\_supervisor\_access\_rights** **MPU\_HAL\_GetM2SupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)  
*Gets M2 access permission in supervisor mode.*

## MPU HAL driver

- static `mpu_user_access_rights MPU_HAL_GetM2UserAccessRights` (`uint32_t baseAddr, mpu_region_num regionNum`)  
*Gets M2 access permission in supervisor mode.*
- static `void MPU_HAL_SetM2SupervisorAccessRights` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_supervisor_access_rights accessRights`)  
*Gets M2 access permission in user mode.*
- static `void MPU_HAL_SetM2UserAccessRights` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_user_access_rights accessRights`)  
*Sets M2 access permission in supervisor mode.*
- static `bool MPU_HAL_IsM2ProcessIdentifierEnabled` (`uint32_t baseAddr, mpu_region_num regionNum`)  
*Sets M2 access permission in user mode.*
- static `void MPU_HAL_SetM2ProcessIdentifierValue` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_process_identifier_value identifierValue`)  
*Checks whether the M2 process identifier enabled in region hit evaluation.*
- static `mpu_supervisor_access_rights MPU_HAL_GetM3SupervisorAccessRights` (`uint32_t baseAddr, mpu_region_num regionNum`)  
*Sets the M2 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*
- static `void MPU_HAL_SetM3SupervisorAccessRights` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_supervisor_access_rights accessRights`)  
*Gets M3 access permission in supervisor mode.*
- static `void MPU_HAL_SetM3UserAccessRights` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_user_access_rights accessRights`)  
*Gets M3 access permission in user mode.*
- static `void MPU_HAL_SetM3ProcessIdentifierEnabled` (`uint32_t baseAddr, mpu_region_num regionNum`)  
*Sets M3 access permission in supervisor mode.*
- static `void MPU_HAL_SetM3ProcessIdentifierValue` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_process_identifier_value identifierValue`)  
*Checks whether the M3 process identifier enabled in region hit evaluation.*
- static `mpu_access_control MPU_HAL_GetM4AccessControl` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_access_type accessType`)  
*Sets M3 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*
- static `void MPU_HAL_SetM4AccessControl` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_access_type accessType, mpu_access_control accessControl`)  
*Gets the M4 access permission.*
- static `void MPU_HAL_SetM5AccessControl` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_access_type accessType, mpu_access_control accessControl`)  
*Sets the M4 access permission.*
- static `void MPU_HAL_SetM6AccessControl` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_access_type accessType, mpu_access_control accessControl`)  
*Gets the M5 access permission.*
- static `void MPU_HAL_SetM7AccessControl` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_access_type accessType, mpu_access_control accessControl`)  
*Sets the M5 access permission.*
- static `void MPU_HAL_SetM8AccessControl` (`uint32_t baseAddr, mpu_region_num regionNum, mpu_access_type accessType, mpu_access_control accessControl`)  
*Gets the M6 access permission.*

- static void **MPU\_HAL\_SetM6AccessControl** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_access\_type** accessType, **mpu\_access\_control** accessControl)
 

*Sets the M6 access permission.*
- static **mpu\_access\_control** **MPU\_HAL\_GetM7AccessControl** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_access\_type** accessType)
 

*Gets the M7 access permission.*
- static void **MPU\_HAL\_SetM7AccessControl** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_access\_type** accessType, **mpu\_access\_control** accessControl)
 

*Sets the M7 access permission.*
- static bool **MPU\_HAL\_IsRegionValid** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Checks whether region is valid.*
- static void **MPU\_HAL\_SetRegionValidValue** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_region\_valid** validValue)
 

*Sets the region valid value.*
- static uint8\_t **MPU\_HAL\_GetProcessIdentifierMask** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Gets the process identifier mask.*
- static void **MPU\_HAL\_SetPIDMASK** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, uint8\_t processIdentifierMask)
 

*Sets the process identifier mask.*
- static uint8\_t **MPU\_HAL\_GetProcessIdentifier** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Gets the process identifier.*
- static void **MPU\_HAL\_SetProcessIdentifier** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, uint8\_t processIdentifier)
 

*Sets the process identifier.*
- static uint32\_t **MPU\_HAL\_GetAllMastersAlternateAcessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Gets all masters access permission from alternative register.*
- static void **MPU\_HAL\_SetAllMastersAlternateAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, uint32\_t accessRights)
 

*Sets all masters access permission through alternative register.*
- static **mpu\_supervisor\_access\_rights** **MPU\_HAL\_GetM0AlternateSupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Gets the M0 access rights in supervisor mode.*
- static **mpu\_user\_access\_rights** **MPU\_HAL\_GetM0AlternateUserAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Gets the M0 access rights in user mode.*
- static void **MPU\_HAL\_SetM0AlternateSupervisorAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_supervisor\_access\_rights** accessRights)
 

*Sets the M0 access rights in supervisor mode.*
- static void **MPU\_HAL\_SetM0AlternateUserAccessRights** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_user\_access\_rights** accessRights)
 

*Sets the M0 access rights in user mode.*
- static bool **MPU\_HAL\_IsM0AlternateProcessIdentifierEnabled** (uint32\_t baseAddr, **mpu\_region\_num** regionNum)
 

*Checks whether the M0 process identifier works in region hit evaluation.*
- static void **MPU\_HAL\_SetM0AlternateProcessIdentifierValue** (uint32\_t baseAddr, **mpu\_region\_num** regionNum, **mpu\_process\_identifier\_value** identifierValue)
 

*Sets the M0 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*

## MPU HAL driver

- static `mpu_supervisor_access_rights MPU_HAL_GetM1AlternateSupervisorAccessRights (uint32_t baseAddr, mpu_region_num regionNum)`  
*Gets M1 access rights in supervisor mode.*
- static `mpu_user_access_rights MPU_HAL_GetM1AlternateUserAccessRights (uint32_t baseAddr, mpu_region_num regionNum)`  
*Gets M1 access rights in user mode.*
- static void `MPU_HAL_SetM1AlternateSupervisorAccessRights (uint32_t baseAddr, mpu_region_num regionNum, mpu_supervisor_access_rights accessRights)`  
*Sets M1 access rights in supervisor mode.*
- static void `MPU_HAL_SetM1AlternateUserAccessRights (uint32_t baseAddr, mpu_region_num regionNum, mpu_user_access_rights accessRights)`  
*Sets M1 access rights in user mode.*
- static bool `MPU_HAL_IsM1AlternateProcessIdentifierEnabled (uint32_t baseAddr, mpu_region_num regionNum)`  
*Checks whether the M1 process identifier works in region hit evaluation.*
- static void `MPU_HAL_SetM1AlternateProcessIdentifierValue (uint32_t baseAddr, mpu_region_num regionNum, mpu_process_identifier_value identifierValue)`  
*Sets M1 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*
- static `mpu_supervisor_access_rights MPU_HAL_GetM2AlternateSupervisorAccessRights (uint32_t baseAddr, mpu_region_num regionNum)`  
*Gets M2 access rights in supervisor mode.*
- static `mpu_user_access_rights MPU_HAL_GetM2AlternateUserAccessRights (uint32_t baseAddr, mpu_region_num regionNum)`  
*Gets the M2 access rights in user mode.*
- static void `MPU_HAL_SetM2AlternateSupervisorAccessRights (uint32_t baseAddr, mpu_region_num regionNum, mpu_supervisor_access_rights accessRights)`  
*Sets M2 access rights in supervisor mode.*
- static void `MPU_HAL_SetM2AlternateUserAccessRights (uint32_t baseAddr, mpu_region_num regionNum, mpu_user_access_rights accessRights)`  
*Sets M2 access rights in user mode.*
- static bool `MPU_HAL_IsM2AlternateProcessIdentifierEnabled (uint32_t baseAddr, mpu_region_num regionNum)`  
*Checks whether the M2 process identifier works in region hit evaluation.*
- static void `MPU_HAL_SetM2AlternateProcessIdentifierValue (uint32_t baseAddr, mpu_region_num regionNum, mpu_process_identifier_value identifierValue)`  
*Sets M2 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*
- static `mpu_supervisor_access_rights MPU_HAL_GetM3AlternateSupervisorAccessRights (uint32_t baseAddr, mpu_region_num regionNum)`  
*Gets M3 access rights in supervisor mode.*
- static `mpu_user_access_rights MPU_HAL_GetM3AlternateUserAccessRights (uint32_t baseAddr, mpu_region_num regionNum)`  
*Gets M3 access rights in user mode.*
- static void `MPU_HAL_SetM3AlternateSupervisorAccessRights (uint32_t baseAddr, mpu_region_num regionNum, mpu_supervisor_access_rights accessRights)`  
*Sets M3 access rights in supervisor mode.*
- static void `MPU_HAL_SetM3AlternateUserAccessRights (uint32_t baseAddr, mpu_region_num regionNum, mpu_user_access_rights accessRights)`  
*Sets M3 access rights in user mode.*
- static bool `MPU_HAL_IsM3AlternateProcessIdentifierEnabled (uint32_t baseAddr, mpu_region_num regionNum)`  
*Checks whether the M3 process identifier works in region hit evaluation.*

`num regionNum)`

*Checks whether the M3 process identifier works in region hit evaluation.*

- static void `MPU_HAL_SetM3AlternateProcessIdentifierValue` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_process_identifier_value` identifierValue)

*Sets M3 process identifier value—1 enable process identifier in region hit evaluation and 0 disable.*

- static `mpu_access_control` `MPU_HAL_GetM4AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType)

*Gets M4 access permission from alternate register.*

- static void `MPU_HAL_SetM4AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType, `mpu_access_control` accessControl)

*Sets M4 access permission through alternate register.*

- static `mpu_access_control` `MPU_HAL_GetM5AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType)

*Gets M5 access permission from alternate register.*

- static void `MPU_HAL_SetM5AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType, `mpu_access_control` accessControl)

*Sets M5 access permission through alternate register.*

- static `mpu_access_control` `MPU_HAL_GetM6AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType)

*Gets M6 access permission from alternate register.*

- static void `MPU_HAL_SetM6AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType, `mpu_access_control` accessControl)

*Sets M6 access permission through alternate register.*

- static `mpu_access_control` `MPU_HAL_GetM7AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType)

*Gets M7 access permission from alternate register.*

- static void `MPU_HAL_SetM7AlternateAccessRights` (uint32\_t baseAddr, `mpu_region_num` regionNum, `mpu_access_type` accessType, `mpu_access_control` accessControl)

*Sets M7 access permission through alternate register.*

- void `MPU_HAL_Init` (uint32\_t baseAddr)

*Initializes the MPU module.*

## 16.1.1 Enumeration Type Documentation

### 16.1.1.1 enum `mpu_region_num`

Enumerator

- |                              |                      |
|------------------------------|----------------------|
| <code>kMPURegionNum00</code> | MPU region number 0. |
| <code>kMPURegionNum01</code> | MPU region number 1. |
| <code>kMPURegionNum02</code> | MPU region number 2. |
| <code>kMPURegionNum03</code> | MPU region number 3. |
| <code>kMPURegionNum04</code> | MPU region number 4. |
| <code>kMPURegionNum05</code> | MPU region number 5. |
| <code>kMPURegionNum06</code> | MPU region number 6. |
| <code>kMPURegionNum07</code> | MPU region number 7. |
| <code>kMPURegionNum08</code> | MPU region number 8. |

## MPU HAL driver

*kMPURegionNum09* MPU region number 9.  
*kMPURegionNum10* MPU region number 10.  
*kMPURegionNum11* MPU region number 11.

### 16.1.1.2 enum mpu\_error\_addr\_reg

Enumerator

*kMPUErrorAddrReg00* MPU error address register 0.  
*kMPUErrorAddrReg01* MPU error address register 1.  
*kMPUErrorAddrReg02* MPU error address register 2.  
*kMPUErrorAddrReg03* MPU error address register 3.  
*kMPUErrorAddrReg04* MPU error address register 4.

### 16.1.1.3 enum mpu\_error\_detail\_reg

Enumerator

*kMPUErrorDetailReg00* MPU error detail register 0.  
*kMPUErrorDetailReg01* MPU error detail register 1.  
*kMPUErrorDetailReg02* MPU error detail register 2.  
*kMPUErrorDetailReg03* MPU error detail register 3.  
*kMPUErrorDetailReg04* MPU error detail register 4.

### 16.1.1.4 enum mpu\_error\_access\_type

Enumerator

*kMPUReadErrorType* MPU error type—read.  
*kMPUWriteErrorType* MPU error type—write.

### 16.1.1.5 enum mpu\_error\_attributes

Enumerator

*kMPUUserModeInstructionAccess* access instruction error in user mode  
*kMPUUserModeDataAccess* access data error in user mode  
*kMPUSupervisorModeInstructionAccess* access instruction error in supervisor mode  
*kMPUSupervisorModeDataAccess* access data error in supervisor mode

### 16.1.1.6 enum mpu\_access\_mode

Enumerator

*kMPUAccessInUserMode* access data or instruction in user mode

*kMPUAccessInSupervisorMode* access data or instruction in supervisor mode

### 16.1.1.7 enum mpu\_master\_num

Enumerator

*kMPUMaster00* Core.

*kMPUMaster01* Debugger.

*kMPUMaster02* DMA.

*kMPUMaster03* ENET.

*kMPUMaster04* USB.

*kMPUMaster05* SDHC.

*kMPUMaster06* undefined.

*kMPUMaster07* undefined.

### 16.1.1.8 enum mpu\_error\_access\_control

Enumerator

*kMPUNoRegionHitError* no region hit error

*kMPUNoneOverlappRegionError* access single region error

*kMPUOverlappRegionError* access overlapping region error

### 16.1.1.9 enum mpu\_supervisor\_access\_rights

Enumerator

*kMPUSupervisorReadWriteExecute* R W E allowed in supervisor mode.

*kMPUSupervisorReadExecute* R E allowed in supervisor mode.

*kMPUSupervisorReadWrite* R W allowed in supervisor mode.

*kMPUSupervisorEqualToUsermode* access permission equal to user mode

### 16.1.1.10 enum mpu\_user\_access\_rights

Enumerator

*kMPUUserNoAccessRights* no access allowed in user mode

*kMPUUserExecute* E allowed in user mode.

## MPU HAL driver

*kMPUUserWrite* W allowed in user mode.

*kMPUUserWriteExecute* W E allowed in user mode.

*kMPUUserRead* R allowed in user mode.

*kMPUUserReadExecute* R E allowed in user mode.

*kMPUUserReadWrite* R W allowed in user mode.

*kMPUUserReadWriteExecute* R W E allowed in user mode.

### 16.1.1.11 enum mpu\_process\_identifier\_value

Enumerator

*kMPUIdentifierDisable* processor identifier disable

*kMPUIdentifierEnable* processor identifier enable

### 16.1.1.12 enum mpu\_access\_control

Enumerator

*kMPUAccessDisable* Read or Write not allowed.

*kMPUAccessEnable* Read or Write allowed.

### 16.1.1.13 enum mpu\_access\_type

Enumerator

*kMPUAccessRead* Access type is read.

*kMPUAccessWrite* Access type is write.

### 16.1.1.14 enum mpu\_region\_valid

Enumerator

*kMPURegionInvalid* region invalid

*kMPURegionValid* region valid

### 16.1.1.15 enum mpu\_status\_t

Enumerator

*kStatus\_MPU\_Success* Succeed.

*kStatus\_MPU\_NotInitialized* MPU is not initialized yet.

*kStatus\_MPU\_NullArgument* Argument is NULL.

## 16.1.2 Function Documentation

16.1.2.1 static void MPU\_HAL\_Enable( uint32\_t *baseAddr* ) [inline], [static]

## MPU HAL driver

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

### 16.1.2.2 static void MPU\_HAL\_Disable ( uint32\_t *baseAddr* ) [inline], [static]

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

### 16.1.2.3 static bool MPU\_HAL\_IsEnabled ( uint32\_t *baseAddr* ) [inline], [static]

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

Return values

|              |                        |
|--------------|------------------------|
| <i>true</i>  | MPU module is enabled  |
| <i>false</i> | MPU module is disabled |

### 16.1.2.4 static uint32\_t MPU\_HAL\_GetNumberOfRegions ( uint32\_t *baseAddr* ) [inline], [static]

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

Return values

|            |                   |
|------------|-------------------|
| <i>the</i> | number of regions |
|------------|-------------------|

### 16.1.2.5 static uint32\_t MPU\_HAL\_GetNumberOfSlaves ( uint32\_t *baseAddr* ) [inline], [static]

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

Return values

|            |                  |
|------------|------------------|
| <i>the</i> | number of slaves |
|------------|------------------|

#### 16.1.2.6 static uint32\_t MPU\_HAL\_GetHardwareRevisionLevel ( uint32\_t *baseAddr* ) [inline], [static]

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

Return values

|                 |                |
|-----------------|----------------|
| <i>hardware</i> | revision level |
|-----------------|----------------|

#### 16.1.2.7 static uint32\_t MPU\_HAL\_GetErrorAccessAddr ( uint32\_t *baseAddr*, mpu\_error\_addr\_reg *regNum* ) [inline], [static]

Parameters

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
| <i>regNum</i>   | Error address register number   |

Return values

|              |                |
|--------------|----------------|
| <i>error</i> | access address |
|--------------|----------------|

#### 16.1.2.8 static uint8\_t MPU\_HAL\_GetErrorSlaveSports ( uint32\_t *baseAddr* ) [inline], [static]

Parameters

## MPU HAL driver

|                 |                                 |
|-----------------|---------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address |
|-----------------|---------------------------------|

Return values

|              |              |
|--------------|--------------|
| <i>error</i> | slave sports |
|--------------|--------------|

**16.1.2.9 static mpu\_error\_access\_type MPU\_HAL\_GetErrorAccessType ( uint32\_t *baseAddr*, mpu\_error\_detail\_reg *errorDetailRegNum* ) [inline], [static]**

Parameters

|                           |                                 |
|---------------------------|---------------------------------|
| <i>baseAddr</i>           | The MPU peripheral base address |
| <i>errorDetail-RegNum</i> | Error detail register number    |

Return values

|              |             |
|--------------|-------------|
| <i>error</i> | access type |
|--------------|-------------|

**16.1.2.10 static mpu\_error\_attributes MPU\_HAL\_GetErrorAttributes ( uint32\_t *baseAddr*, mpu\_error\_detail\_reg *errorDetailRegNum* ) [inline], [static]**

Parameters

|                           |                                 |
|---------------------------|---------------------------------|
| <i>baseAddr</i>           | The MPU peripheral base address |
| <i>errorDetail-RegNum</i> | Detail error register number    |

Return values

|              |                   |
|--------------|-------------------|
| <i>error</i> | access attributes |
|--------------|-------------------|

**16.1.2.11 static mpu\_master\_num MPU\_HAL\_GetErrorMasterNum ( uint32\_t *baseAddr*, mpu\_error\_detail\_reg *errorDetailRegNum* ) [inline], [static]**

Parameters

|                           |                                 |
|---------------------------|---------------------------------|
| <i>baseAddr</i>           | The MPU peripheral base address |
| <i>errorDetail-RegNum</i> | Error register number           |

Return values

|              |               |
|--------------|---------------|
| <i>error</i> | master number |
|--------------|---------------|

#### 16.1.2.12 static uint32\_t MPU\_HAL\_GetErrorProcessIdentifier ( uint32\_t *baseAddr*, mpu\_error\_detail\_reg *errorDetailRegNum* ) [inline], [static]

Parameters

|                           |                                 |
|---------------------------|---------------------------------|
| <i>baseAddr</i>           | The MPU peripheral base address |
| <i>errorDetail-RegNum</i> | Error register number           |

Return values

|              |                    |
|--------------|--------------------|
| <i>error</i> | process identifier |
|--------------|--------------------|

#### 16.1.2.13 static mpu\_error\_access\_control MPU\_HAL\_GetErrorAccessControl ( uint32\_t *baseAddr*, mpu\_error\_detail\_reg *errorDetailRegNum* ) [inline], [static]

Parameters

|                           |                                 |
|---------------------------|---------------------------------|
| <i>baseAddr</i>           | The MPU peripheral base address |
| <i>errorDetail-RegNum</i> | Error register number           |

Return values

|              |                |
|--------------|----------------|
| <i>error</i> | access control |
|--------------|----------------|

#### 16.1.2.14 static uint32\_t MPU\_HAL\_GetRegionStartAddr ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]

## MPU HAL driver

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|               |               |
|---------------|---------------|
| <i>region</i> | start address |
|---------------|---------------|

**16.1.2.15 static void MPU\_HAL\_SetRegionStartAddr ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, uint32\_t *startAddr* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |
| <i>startAddr</i> | Region start address            |

**16.1.2.16 static uint32\_t MPU\_HAL\_GetRegionEndAddr ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|               |             |
|---------------|-------------|
| <i>region</i> | end address |
|---------------|-------------|

**16.1.2.17 static void MPU\_HAL\_SetRegionEndAddr ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, uint32\_t *endAddr* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |
| <i>endAddr</i>   | Region end address              |

**16.1.2.18 static uint32\_t MPU\_HAL\_GetAllMastersAccessRights ( uint32\_t *baseAddr*,  
mpu\_region\_num *regionNum* ) [inline], [static]**

## MPU HAL driver

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|            |                           |
|------------|---------------------------|
| <i>all</i> | masters access permission |
|------------|---------------------------|

**16.1.2.19 static void MPU\_HAL\_SetAllMastersAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, uint32\_t *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | All masters access rights       |

**16.1.2.20 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM0SupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master0</i> | access permission |
|----------------|-------------------|

**16.1.2.21 static mpu\_user\_access\_rights MPU\_HAL\_GetM0UserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master0</i> | access permission |
|----------------|-------------------|

**16.1.2.22 static void MPU\_HAL\_SetM0SupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_supervisor\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master0 access permission       |

**16.1.2.23 static void MPU\_HAL\_SetM0UserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_user\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master0 access permission       |

**16.1.2.24 static bool MPU\_HAL\_IsM0ProcessIdentifierEnabled ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

## MPU HAL driver

Return values

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m0 process identifier is enabled  |
| <i>false</i> | m0 process identifier is disabled |

**16.1.2.25 static void MPU\_HAL\_SetM0ProcessIdentifierValue ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_process\_identifier\_value *identifierValue* ) [inline], [static]**

Parameters

|                        |                                 |
|------------------------|---------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address |
| <i>regionNum</i>       | MPU region number               |
| <i>identifierValue</i> | Process identifier value        |

**16.1.2.26 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM1SupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master1</i> | access permission |
|----------------|-------------------|

**16.1.2.27 static mpu\_user\_access\_rights MPU\_HAL\_GetM1UserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master1</i> | access permission |
|----------------|-------------------|

**16.1.2.28 static void MPU\_HAL\_SetM1SupervisorAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_supervisor\_access\_rights accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master1 access permission       |

**16.1.2.29 static void MPU\_HAL\_SetM1UserAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_user\_access\_rights accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master1 access permission       |

**16.1.2.30 static bool MPU\_HAL\_IsM1ProcessIdentifierEnabled ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

---

## MPU HAL driver

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m1 process identifier is enabled  |
| <i>false</i> | m1 process identifier is disabled |

**16.1.2.31 static void MPU\_HAL\_SetM1ProcessIdentifierValue ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_process\_identifier\_value identifierValue* ) [inline], [static]**

Parameters

|                        |                                 |
|------------------------|---------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address |
| <i>regionNum</i>       | MPU region number               |
| <i>identifierValue</i> | Process identifier value        |

**16.1.2.32 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM2SupervisorAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master2</i> | access permission |
|----------------|-------------------|

**16.1.2.33 static mpu\_user\_access\_rights MPU\_HAL\_GetM2UserAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master2</i> | access permission |
|----------------|-------------------|

**16.1.2.34 static void MPU\_HAL\_SetM2SupervisorAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_supervisor\_access\_rights accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master2 access permission       |

**16.1.2.35 static void MPU\_HAL\_SetM2UserAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_user\_access\_rights accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master2 access permission       |

**16.1.2.36 static bool MPU\_HAL\_IsM2ProcessIdentifierEnabled ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

---

## MPU HAL driver

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m2 process identifier is enabled  |
| <i>false</i> | m2 process identifier is disabled |

**16.1.2.37 static void MPU\_HAL\_SetM2ProcessIdentifierValue ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_process\_identifier\_value identifierValue* ) [inline], [static]**

Parameters

|                        |                                  |
|------------------------|----------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address. |
| <i>regionNum</i>       | MPU region number.               |
| <i>identifierValue</i> | Process identifier value.        |

**16.1.2.38 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM3SupervisorAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master3</i> | access permission |
|----------------|-------------------|

**16.1.2.39 static mpu\_user\_access\_rights MPU\_HAL\_GetM3UserAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master3</i> | access permission |
|----------------|-------------------|

**16.1.2.40 static void MPU\_HAL\_SetM3SupervisorAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_supervisor\_access\_rights accessRights* ) [inline], [static]**

Parameters

|                     |                                  |
|---------------------|----------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address. |
| <i>regionNum</i>    | MPU region number.               |
| <i>accessRights</i> | Master3 access permission.       |

**16.1.2.41 static void MPU\_HAL\_SetM3UserAccessRights ( *uint32\_t baseAddr, mpu\_region\_num regionNum, mpu\_user\_access\_rights accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master3 access permission       |

**16.1.2.42 static bool MPU\_HAL\_IsM3ProcessIdentifierEnabled ( *uint32\_t baseAddr, mpu\_region\_num regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m3 process identifier is enabled  |
| <i>false</i> | m3 process identifier is disabled |

**16.1.2.43 static void MPU\_HAL\_SetM3ProcessIdentifierValue ( uint32\_t *baseAddr*,  
mpu\_region\_num *regionNum*, mpu\_process\_identifier\_value *identifierValue* )  
[inline], [static]**

Parameters

|                        |                                 |
|------------------------|---------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address |
| <i>regionNum</i>       | MPU region number               |
| <i>identifierValue</i> | Process identifier value        |

**16.1.2.44 static mpu\_access\_control MPU\_HAL\_GetM4AccessControl ( uint32\_t  
*baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* )  
[inline], [static]**

Parameters

|                   |                                 |
|-------------------|---------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address |
| <i>regionNum</i>  | MPU region number               |
| <i>accessType</i> | Access type Read/Write          |

Return values

|             |                     |
|-------------|---------------------|
| <i>read</i> | or write permission |
|-------------|---------------------|

**16.1.2.45 static void MPU\_HAL\_SetM4AccessControl ( uint32\_t *baseAddr*,  
mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*,  
mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                 |
|----------------------|---------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address |
| <i>regionNum</i>     | MPU region number               |
| <i>accessType</i>    | Access type Read/Write          |
| <i>accessControl</i> | Access permission               |

**16.1.2.46 static mpu\_access\_control MPU\_HAL\_GetM5AccessControl ( uint32\_t  
*baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* )  
[inline], [static]**

## MPU HAL driver

Parameters

|                   |                                 |
|-------------------|---------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address |
| <i>regionNum</i>  | MPU region number               |
| <i>accessType</i> | Access type Read/Write          |

Return values

|             |                     |
|-------------|---------------------|
| <i>read</i> | or write permission |
|-------------|---------------------|

**16.1.2.47 static void MPU\_HAL\_SetM5AccessControl ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*, mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                 |
|----------------------|---------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address |
| <i>regionNum</i>     | MPU region number               |
| <i>accessType</i>    | Access type Read/Write          |
| <i>accessControl</i> | Access permission               |

**16.1.2.48 static mpu\_access\_control MPU\_HAL\_GetM6AccessControl ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* ) [inline], [static]**

Parameters

|                   |                                 |
|-------------------|---------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address |
| <i>regionNum</i>  | MPU region number               |
| <i>accessType</i> | access type Read/Write          |

Return values

|             |                     |
|-------------|---------------------|
| <i>read</i> | or write permission |
|-------------|---------------------|

**16.1.2.49 static void MPU\_HAL\_SetM6AccessControl ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*, mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                 |
|----------------------|---------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address |
| <i>regionNum</i>     | MPU region number               |
| <i>accessType</i>    | Access type Read/Write          |
| <i>accessControl</i> | Access permission               |

**16.1.2.50 static mpu\_access\_control MPU\_HAL\_GetM7AccessControl ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* ) [inline], [static]**

Parameters

|                   |                                 |
|-------------------|---------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address |
| <i>regionNum</i>  | MPU region number               |
| <i>accessType</i> | Access type Read/Write          |

Return values

|             |                     |
|-------------|---------------------|
| <i>read</i> | or write permission |
|-------------|---------------------|

**16.1.2.51 static void MPU\_HAL\_SetM7AccessControl ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*, mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                 |
|----------------------|---------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address |
| <i>regionNum</i>     | MPU region number               |
| <i>accessType</i>    | Access type Read/Write          |
| <i>accessControl</i> | Access permission               |

**16.1.2.52 static bool MPU\_HAL\_IsRegionValid ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

## MPU HAL driver

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|              |                   |
|--------------|-------------------|
| <i>true</i>  | region is valid   |
| <i>false</i> | region is invalid |

**16.1.2.53 static void MPU\_HAL\_SetRegionValidValue ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_region\_valid *validValue* ) [inline], [static]**

Parameters

|                   |                                 |
|-------------------|---------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address |
| <i>regionNum</i>  | MPU region number               |
| <i>validValue</i> | Region valid value              |

**16.1.2.54 static uint8\_t MPU\_HAL\_GetProcessIdentifierMask ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|               |                         |
|---------------|-------------------------|
| <i>region</i> | process identifier mask |
|---------------|-------------------------|

**16.1.2.55 static void MPU\_HAL\_SetPIDMASK ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, uint8\_t *processIdentifierMask* ) [inline], [static]**

Parameters

|                                    |                                 |
|------------------------------------|---------------------------------|
| <i>baseAddr</i>                    | The MPU peripheral base address |
| <i>regionNum</i>                   | MPU region number               |
| <i>process-<br/>IdentifierMask</i> | Process identifier mask value   |

**16.1.2.56 static uint8\_t MPU\_HAL\_GetProcessIdentifier ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |            |
|----------------|------------|
| <i>process</i> | identifier |
|----------------|------------|

**16.1.2.57 static void MPU\_HAL\_SetProcessIdentifier ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, uint8\_t *processIdentifier* ) [inline], [static]**

Parameters

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>baseAddr</i>                | The MPU peripheral base address |
| <i>regionNum</i>               | MPU region number               |
| <i>process-<br/>Identifier</i> | Process identifier              |

**16.1.2.58 static uint32\_t MPU\_HAL\_GetAllMastersAlternateAcessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

## MPU HAL driver

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|            |                           |
|------------|---------------------------|
| <i>all</i> | masters access permission |
|------------|---------------------------|

**16.1.2.59 static void MPU\_HAL\_SetAllMastersAlternateAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, uint32\_t *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | All masters access permission   |

**16.1.2.60 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM0AlternateSupervisor-AccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master0</i> | access permission |
|----------------|-------------------|

**16.1.2.61 static mpu\_user\_access\_rights MPU\_HAL\_GetM0AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master0</i> | access permission |
|----------------|-------------------|

**16.1.2.62 static void MPU\_HAL\_SetM0AlternateSupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_supervisor\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master0 access permission       |

**16.1.2.63 static void MPU\_HAL\_SetM0AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_user\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master0 access permission       |

**16.1.2.64 static bool MPU\_HAL\_IsM0AlternateProcessIdentifierEnabled ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

## MPU HAL driver

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m0 process identifier is enabled  |
| <i>false</i> | m0 process identifier is disabled |

**16.1.2.65 static void MPU\_HAL\_SetM0AlternateProcessIdentifierValue ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_process\_identifier\_value *identifierValue* ) [inline], [static]**

Parameters

|                        |                                 |
|------------------------|---------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address |
| <i>regionNum</i>       | MPU region number               |
| <i>identifierValue</i> | Process identifier value        |

**16.1.2.66 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM1AlternateSupervisor-AccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master1</i> | access permission |
|----------------|-------------------|

**16.1.2.67 static mpu\_user\_access\_rights MPU\_HAL\_GetM1AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|                |                   |
|----------------|-------------------|
| <i>Master1</i> | access permission |
|----------------|-------------------|

**16.1.2.68 static void MPU\_HAL\_SetM1AlternateSupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_supervisor\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master1 access permission       |

**16.1.2.69 static void MPU\_HAL\_SetM1AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_user\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master1 access permission       |

**16.1.2.70 static bool MPU\_HAL\_IsM1AlternateProcessIdentifierEnabled ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m1 process identifier is enabled  |
| <i>false</i> | m1 process identifier is disabled |

**16.1.2.71 static void MPU\_HAL\_SetM1AlternateProcessIdentifierValue ( uint32\_t  
baseAddr, mpu\_region\_num *regionNum*, mpu\_process\_identifier\_value  
*identifierValue* ) [inline], [static]**

Parameters

|                        |                                 |
|------------------------|---------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address |
| <i>regionNum</i>       | MPU region number               |
| <i>identifierValue</i> | process identifier value        |

**16.1.2.72 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM2AlternateSupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|           |                   |
|-----------|-------------------|
| <i>M2</i> | access permission |
|-----------|-------------------|

**16.1.2.73 static mpu\_user\_access\_rights MPU\_HAL\_GetM2AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|           |                   |
|-----------|-------------------|
| <i>M2</i> | access permission |
|-----------|-------------------|

**16.1.2.74 static void MPU\_HAL\_SetM2AlternateSupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_supervisor\_access\_rights *accessRights* ) [inline], [static]**

## MPU HAL driver

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | M2 access permission            |

**16.1.2.75 static void MPU\_HAL\_SetM2AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_user\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | M2 access permission            |

**16.1.2.76 static bool MPU\_HAL\_IsM2AlternateProcessIdentifierEnabled ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|              |                                   |
|--------------|-----------------------------------|
| <i>true</i>  | m2 process identifier is enabled  |
| <i>false</i> | m2 process identifier is disabled |

**16.1.2.77 static void MPU\_HAL\_SetM2AlternateProcessIdentifierValue ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_process\_identifier\_value *identifierValue* ) [inline], [static]**

Parameters

|                        |                                 |
|------------------------|---------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address |
| <i>regionNum</i>       | MPU region number               |
| <i>identifierValue</i> | process identifier value        |

**16.1.2.78 static mpu\_supervisor\_access\_rights MPU\_HAL\_GetM3AlternateSupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|           |                   |
|-----------|-------------------|
| <i>M3</i> | access permission |
|-----------|-------------------|

**16.1.2.79 static mpu\_user\_access\_rights MPU\_HAL\_GetM3AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                 |
|------------------|---------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address |
| <i>regionNum</i> | MPU region number               |

Return values

|           |                   |
|-----------|-------------------|
| <i>M3</i> | access permission |
|-----------|-------------------|

**16.1.2.80 static void MPU\_HAL\_SetM3AlternateSupervisorAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_supervisor\_access\_rights *accessRights* ) [inline], [static]**

## MPU HAL driver

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master3 access permission       |

**16.1.2.81 static void MPU\_HAL\_SetM3AlternateUserAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_user\_access\_rights *accessRights* ) [inline], [static]**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>baseAddr</i>     | The MPU peripheral base address |
| <i>regionNum</i>    | MPU region number               |
| <i>accessRights</i> | Master3 access permission       |

**16.1.2.82 static bool MPU\_HAL\_IsM3AlternateProcessIdentifierEnabled ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum* ) [inline], [static]**

Parameters

|                  |                                  |
|------------------|----------------------------------|
| <i>baseAddr</i>  | The MPU peripheral base address. |
| <i>regionNum</i> | MPU region number.               |

Return values

|              |                                    |
|--------------|------------------------------------|
| <i>true</i>  | m3 process identifier is enabled.  |
| <i>false</i> | m3 process identifier is disabled. |

**16.1.2.83 static void MPU\_HAL\_SetM3AlternateProcessIdentifierValue ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_process\_identifier\_value *identifierValue* ) [inline], [static]**

Parameters

|                        |                                  |
|------------------------|----------------------------------|
| <i>baseAddr</i>        | The MPU peripheral base address. |
| <i>regionNum</i>       | MPU region number.               |
| <i>identifierValue</i> | process identifier value.        |

**16.1.2.84 static mpu\_access\_control MPU\_HAL\_GetM4AlternateAccessRights ( uint32\_t  
*baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* )  
[inline], [static]**

Parameters

|                   |                                  |
|-------------------|----------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address. |
| <i>regionNum</i>  | MPU region number.               |
| <i>accessType</i> | Access type Read/Write.          |

Return values

|             |                      |
|-------------|----------------------|
| <i>read</i> | or write permission. |
|-------------|----------------------|

**16.1.2.85 static void MPU\_HAL\_SetM4AlternateAccessRights ( uint32\_t  
*baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*,  
mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                  |
|----------------------|----------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address. |
| <i>regionNum</i>     | MPU region number.               |
| <i>accessType</i>    | Access type Read/Write.          |
| <i>accessControl</i> | Access permission.               |

**16.1.2.86 static mpu\_access\_control MPU\_HAL\_GetM5AlternateAccessRights ( uint32\_t  
*baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* )  
[inline], [static]**

## MPU HAL driver

Parameters

|                   |                                  |
|-------------------|----------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address. |
| <i>regionNum</i>  | MPU region number.               |
| <i>accessType</i> | Access type Read/Write.          |

Return values

|             |                      |
|-------------|----------------------|
| <i>read</i> | or write permission. |
|-------------|----------------------|

**16.1.2.87 static void MPU\_HAL\_SetM5AlternateAccessRights ( uint32\_t  
                  *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*,  
                  mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                  |
|----------------------|----------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address. |
| <i>regionNum</i>     | MPU region number.               |
| <i>accessType</i>    | Access type Read/Write.          |
| <i>accessControl</i> | Master5 Access permission.       |

**16.1.2.88 static mpu\_access\_control MPU\_HAL\_GetM6AlternateAccessRights ( uint32\_t  
                  *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* )  
                  [inline], [static]**

Parameters

|                   |                                  |
|-------------------|----------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address. |
| <i>regionNum</i>  | MPU region number.               |
| <i>accessType</i> | Access type Read/Write.          |

Return values

|             |                      |
|-------------|----------------------|
| <i>read</i> | or write permission. |
|-------------|----------------------|

**16.1.2.89 static void MPU\_HAL\_SetM6AlternateAccessRights ( uint32\_t  
                  *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*,  
                  mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                  |
|----------------------|----------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address. |
| <i>regionNum</i>     | MPU region number.               |
| <i>accessType</i>    | Access type Read/Write.          |
| <i>accessControl</i> | Master6 access permission.       |

**16.1.2.90 static mpu\_access\_control MPU\_HAL\_GetM7AlternateAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType* ) [inline], [static]**

Parameters

|                   |                                  |
|-------------------|----------------------------------|
| <i>baseAddr</i>   | The MPU peripheral base address. |
| <i>regionNum</i>  | MPU region number.               |
| <i>accessType</i> | Access type Read/Write.          |

Return values

|             |                      |
|-------------|----------------------|
| <i>read</i> | or write permission. |
|-------------|----------------------|

**16.1.2.91 static void MPU\_HAL\_SetM7AlternateAccessRights ( uint32\_t *baseAddr*, mpu\_region\_num *regionNum*, mpu\_access\_type *accessType*, mpu\_access\_control *accessControl* ) [inline], [static]**

Parameters

|                      |                                  |
|----------------------|----------------------------------|
| <i>baseAddr</i>      | The MPU peripheral base address. |
| <i>regionNum</i>     | MPU region number.               |
| <i>accessType</i>    | Access type Read/Write.          |
| <i>accessControl</i> | Master7 access permission.       |

**16.1.2.92 void MPU\_HAL\_Init ( uint32\_t *baseAddr* )**

## MPU HAL driver

Parameters

|                 |                                  |
|-----------------|----------------------------------|
| <i>baseAddr</i> | The MPU peripheral base address. |
|-----------------|----------------------------------|

## 16.2 MPU Peripheral Driver

This chapter describes the programming interface of the MPU Peripheral driver.

### Data Structures

- struct `mpu_access_rights_t`  
*Data c for the MPU region access permission initialize. [More...](#)*
- struct `mpu_region_config_t`  
*Data v for MPU region initialize. [More...](#)*
- struct `mpu_user_config_t`  
*Data The chapter describes the programming interface of the for MPU region initialize. [More...](#)*
- struct `mpu_state_t`  
*MPU driver user call back function. [More...](#)*

### TypeDefs

- typedef void(\* `mpu_callback_t` )(void)  
*Define type of user-defined callback function.*

### MPU Driver

- `mpu_status_t MPU_DRV_Init` (uint32\_t instance, `mpu_user_config_t` \*userConfigPtr, `mpu_state_t` \*userStatePtr)  
*Initializes the MPU driver.*
- `mpu_status_t MPU_DRV_RegionInit` (uint32\_t instance, `mpu_region_config_t` \*regionConfigPtr)  
*Initializes the MPU region.*
- `mpu_status_t MPU_DRV_Deinit` (uint32\_t instance)  
*Deinitializes the MPU region.*
- `mpu_status_t MPU_DRV_SetRegionAccessPermission` (uint32\_t instance, `mpu_region_num` regionNum, `mpu_access_rights_t` accessRights)  
*Configures the MPU region access permission.*
- `mpu_access_rights_t MPU_DRV_GetRegionAccessPermission` (uint32\_t instance, `mpu_region_num` regionNum)  
*Gets the MPU region access permission.*
- `bool MPU_DRV_IsRegionValid` (uint32\_t instance, `mpu_region_num` regionNum)  
*Checks whether the MPU region is valid.*
- `mpu_status_t MPU_DRV_SetRegionValid` (uint32\_t instance, `mpu_region_num` regionNum)  
*Sets the MPU region valid.*
- `mpu_status_t MPU_DRV_InstallCallback` (uint32\_t instance, `mpu_callback_t` userCallback)  
*Installs the user-defined callback in MPU module.*
- `void MPU_DRV_IRQHandler` (uint32\_t instance)  
*Driver-defined ISR in MPU module.*

## MPU Peripheral Driver

### 16.2.0.93 MPU Peripheral Driver

#### Overview

The MPU driver provides an easy method to configure MPU.

#### Initialization

To initialize the MPU module, call the [MPU\\_DRV\\_Init\(\)](#) function and provide the user configuration data structure. This function sets the configuration of the MPU module automatically and enable MPU module.

Attention: The configuration start address, end address, region valid value and the debugger's access permission of MPU region 0 cannot be changed.

This is example code for configuring MPU driver:

```
/* Define MPU memory access permission configuration structure . */
struct mpu_access_rights_t mpuAccessRights{
 .m0UserMode = kMPUUserNoAccessRights;
 .m0SupervisorMode = kMPUSupervisorReadWriteExecute;
 .m0Process_identifier = kMPUIIdentifierDisable;
 .m1UserMode = kMPUUserNoAccessRights;
 .m1SupervisorMode = kMPUSupervisorEqualToUsermode;
 .m1Process_identifier = kMPUIIdentifierDisable;
 .m2UserMode = kMPUUserNoAccessRights;
 .m2SupervisorMode = kMPUSupervisorEqualToUsermode;
 .m2Process_identifier = kMPUIIdentifierDisable;
 .m3UserMode = kMPUUserNoAccessRights;
 .m3SupervisorMode = kMPUSupervisorEqualToUsermode;
 .m3Process_identifier = kMPUIIdentifierDisable;
 .m4WriteControl = kMPUAccessDisable;
 .m4ReadControl = kMPUAccessDisable;
 .m5WriteControl = kMPUAccessDisable;
 .m5ReadControl = kMPUAccessDisable;
 .m6WriteControl = kMPUAccessDisable;
 .m6ReadControl = kMPUAccessDisable;
 .m7WriteControl = kMPUAccessDisable;
 .m7ReadControl = kMPUAccessDisable;
};

/* Define MPU region configuration structure . */
struct mpu_region_config_t mpuRegionConfig{
 .regionNum = kMPURegionNum00;
 .startAddr = 0x0;
 .endAddr = 0xffffffff;
 .accessRights = mpuAccessRights;
};

/* Define MPU user configuration structure . */
struct mpu_user_config_t mpuUserConfig{
 .regionConfig = mpuRegionConfig;
 .next = NULL;
}mpu_user_config_t;

/* Initialize MPU region0. */
MPU_DRV_Init(0, &mpuUserConfig);
```

## MPU Interrupt

1. The interrupt corresponding to BUSFAULT which causes an error by accessing core.
2. Define MPU IRQ function

```
void MPU_DRV_IRQHandler(uint32_t instance)
{
 assert(instance < HW_MPU_INSTANCE_COUNT);

 if (mpu_state_ptrs[instance])
 {
 if (mpu_state_ptrs[instance]->userCallbackFunc)
 {
 /* Execute user-defined callback function. */
 (* (mpu_state_ptrs[instance]->userCallbackFunc)) ();
 }
 }
}
```

### 16.2.1 Data Structure Documentation

#### 16.2.1.1 struct mpu\_access\_rights\_t

This structure is used when MPU\_DRV\_Init function is called.

#### Data Fields

- uint32\_t m0UserMode: 3  
*master0 access permission in user mode*
- uint32\_t m0SupervisorMode: 2  
*master0 access permission in supervisor mode*
- uint32\_t m0Process\_identifier: 1  
*master0 process identifier enable value*
- uint32\_t m1UserMode: 3  
*master1 access permission in user mode*
- uint32\_t m1SupervisorMode: 2  
*master1 access permission in supervisor mode*
- uint32\_t m1Process\_identifier: 1  
*master1 process identifier enable value*
- uint32\_t m2UserMode: 3  
*master2 access permission in user mode*
- uint32\_t m2SupervisorMode: 2  
*master2 access permission in supervisor mode*
- uint32\_t m2Process\_identifier: 1  
*master2 process identifier enable value*
- uint32\_t m3UserMode: 3  
*master3 access permission in user mode*
- uint32\_t m3SupervisorMode: 2  
*master3 access permission in supervisor mode*
- uint32\_t m3Process\_identifier: 1  
*master3 process identifier enable value*
- uint32\_t m4WriteControl: 1

## MPU Peripheral Driver

- *master4 write access permission*
  - `uint32_t m4ReadControl: 1`
  - *master4 read access permission*
- `uint32_t m5WriteControl: 1`
  - *master5 write access permission*
  - `uint32_t m5ReadControl: 1`
  - *master5 read access permission*
- `uint32_t m6WriteControl: 1`
  - *master6 write access permission*
  - `uint32_t m6ReadControl: 1`
  - *master6 read access permission*
- `uint32_t m7WriteControl: 1`
  - *master7 write access permission*
  - `uint32_t m7ReadControl: 1`
  - *master7 read access permission*

### 16.2.1.2 struct mpu\_region\_config\_t

This structure is used when the MPU\_DRV\_Init function is called.

#### Data Fields

- `mpu_region_num regionNum`  
*MPU region number.*
- `uint32_t startAddr`  
*memory region start address*
- `uint32_t endAddr`  
*memory region end address*
- `mpu_access_rights_t accessRights`  
*all masters access permission*

### 16.2.1.3 struct mpu\_user\_config\_t

This structure is used when the MPU\_DRV\_Init function is called.

#### Data Fields

- `mpu_region_config_t regionConfig`  
*region access permission*
- `struct MpuUserConfig * next`  
*pointer to the next structure*

### 16.2.1.4 struct mpu\_state\_t

The contents of this structure provide a callback function.

## Data Fields

- [mpu\\_callback\\_t userCallbackFunc](#)  
*Callback function that would be executed in ISR.*

### 16.2.1.4.0.39 Field Documentation

#### 16.2.1.4.0.39.1 [mpu\\_callback\\_t mpu\\_state\\_t::userCallbackFunc](#)

## 16.2.2 Function Documentation

### 16.2.2.1 [mpu\\_status\\_t MPU\\_DRV\\_Init \( uint32\\_t instance, mpu\\_user\\_config\\_t \\* userConfigPtr, mpu\\_state\\_t \\* userStatePtr \)](#)

Parameters

|                      |                                                                                          |
|----------------------|------------------------------------------------------------------------------------------|
| <i>instance</i>      | The MPU peripheral instance number.                                                      |
| <i>userConfigPtr</i> | The pointer to the MPU user configure structure, see <a href="#">mpu_user_config_t</a> . |
| <i>userStatePtr</i>  | The pointer of run time structure.                                                       |

Return values

|                            |                                          |
|----------------------------|------------------------------------------|
| <i>kStatus_MPU_Success</i> | means success. Otherwise, means failure. |
|----------------------------|------------------------------------------|

### 16.2.2.2 [mpu\\_status\\_t MPU\\_DRV\\_RegionInit \( uint32\\_t instance, mpu\\_region\\_config\\_t \\* regionConfigPtr \)](#)

Parameters

|                              |                                                                                          |
|------------------------------|------------------------------------------------------------------------------------------|
| <i>instance</i>              | The MPU peripheral instance number.                                                      |
| <i>regionConfig-<br/>Ptr</i> | The pointer to the MPU user configure structure, see <a href="#">mpu_user_config_t</a> . |

Return values

|                            |                                          |
|----------------------------|------------------------------------------|
| <i>kStatus_MPU_Success</i> | means success, otherwise, means failure. |
|----------------------------|------------------------------------------|

### 16.2.2.3 [mpu\\_status\\_t MPU\\_DRV\\_Deinit \( uint32\\_t instance \)](#)

## MPU Peripheral Driver

Parameters

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The MPU peripheral instance number. |
|-----------------|-------------------------------------|

Return values

|                            |                                          |
|----------------------------|------------------------------------------|
| <i>kStatus_MPU_Success</i> | means success. Otherwise, means failure. |
|----------------------------|------------------------------------------|

### 16.2.2.4 **mpu\_status\_t MPU\_DRV\_SetRegionAccessPermission ( uint32\_t *instance*, mpu\_region\_num *regionNum*, mpu\_access\_rights\_t *accessRights* )**

Parameters

|                     |                                           |
|---------------------|-------------------------------------------|
| <i>instance</i>     | The MPU peripheral instance number.       |
| <i>regionNum</i>    | The MPU region number.                    |
| <i>accessRights</i> | A pointer to access permission structure. |

Return values

|                            |                                          |
|----------------------------|------------------------------------------|
| <i>kStatus_MPU_Success</i> | means success, otherwise, means failure. |
|----------------------------|------------------------------------------|

### 16.2.2.5 **mpu\_access\_rights\_t MPU\_DRV\_GetRegionAccessPermission ( uint32\_t *instance*, mpu\_region\_num *regionNum* )**

Parameters

|                  |                                     |
|------------------|-------------------------------------|
| <i>instance</i>  | The MPU peripheral instance number. |
| <i>regionNum</i> | The MPU region number.              |

Return values

|               |             |
|---------------|-------------|
| <i>access</i> | permission. |
|---------------|-------------|

### 16.2.2.6 **bool MPU\_DRV\_IsRegionValid ( uint32\_t *instance*, mpu\_region\_num *regionNum* )**

Parameters

|                  |                                     |
|------------------|-------------------------------------|
| <i>instance</i>  | The MPU peripheral instance number. |
| <i>regionNum</i> | MPU region number.                  |

Return values

|             |        |
|-------------|--------|
| <i>bool</i> | value. |
|-------------|--------|

#### 16.2.2.7 **mpu\_status\_t MPU\_DRV\_SetRegionValid ( uint32\_t *instance*, mpu\_region\_num *regionNum* )**

Parameters

|                  |                                     |
|------------------|-------------------------------------|
| <i>instance</i>  | The MPU peripheral instance number. |
| <i>regionNum</i> | MPU region number.                  |

#### 16.2.2.8 **mpu\_status\_t MPU\_DRV\_InstallCallback ( uint32\_t *instance*, mpu\_callback\_t *userCallback* )**

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>instance</i>     | MPU instance ID.                |
| <i>userCallback</i> | User-defined callback function. |

Return values

|                            |                                        |
|----------------------------|----------------------------------------|
| <i>kStatus_MPU_Success</i> | means succeed, otherwise means failed. |
|----------------------------|----------------------------------------|

#### 16.2.2.9 **void MPU\_DRV\_IRQHandler ( uint32\_t *instance* )**

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | MPU instance ID. |
|-----------------|------------------|



# Chapter 17

## Programmable Delay Block (PDB)

The Kinetis SDK provides both HAL and Peripheral drivers for the Programmable Delay Block (PDB) block of Kinetis devices.

### Modules

- [PDB HAL Driver](#)

*This part describes the programming interface of the PDB HAL driver.*

- [PDB Peripheral Driver](#)

*This part describes the programming interface of the PDB Peripheral driver.*

### 17.1 PDB HAL Driver

This chapter describes the programming interface of the PDB HAL driver.

#### Enumerations

- enum  `pdb_status_t {`  
 `kStatus_PDB_Success = 0U,`  
 `kStatus_PDB_InvalidArgument = 1U,`  
 `kStatus_PDB_Failed = 2U }`  
*PDB status return codes.*
- enum  `pdb_load_mode_t {`  
 `kPdbLoadImmediately = 0U,`  
 `kPdbLoadAtModuloCounter = 1U,`  
 `kPdbLoadAtNextTrigger = 2U,`  
 `kPdbLoadAtModuloCounterOrNextTrigger = 3U }`  
*Defines the type of value load mode for the PDB module.*
- enum  `pdb_clk_prescaler_div_mode_t {`  
 `kPdbClkPreDivBy1 = 0U,`  
 `kPdbClkPreDivBy2 = 1U,`  
 `kPdbClkPreDivBy4 = 2U,`  
 `kPdbClkPreDivBy8 = 3U,`  
 `kPdbClkPreDivBy16 = 4U,`  
 `kPdbClkPreDivBy32 = 5U,`  
 `kPdbClkPreDivBy64 = 6U,`  
 `kPdbClkPreDivBy128 = 7U }`  
*Defines the type of prescaler divider for the PDB counter clock.*
- enum  `pdb_trigger_src_mode_t {`  
 `kPdbTrigger0 = 0U,`  
 `kPdbTrigger1 = 1U,`  
 `kPdbTrigger2 = 2U,`  
 `kPdbTrigger3 = 3U,`  
 `kPdbTrigger4 = 4U,`  
 `kPdbTrigger5 = 5U,`  
 `kPdbTrigger6 = 6U,`  
 `kPdbTrigger7 = 7U,`  
 `kPdbTrigger8 = 8U,`  
 `kPdbTrigger9 = 9U,`  
 `kPdbTrigger10 = 10U,`  
 `kPdbTrigger11 = 11U,`  
 `kPdbTrigger12 = 12U,`  
 `kPdbTrigger13 = 13U,`  
 `kPdbTrigger14 = 14U,`  
 `kPdbSoftTrigger = 15U }`  
*Defines the type of trigger source mode for the PDB.*

- enum `pdb_mult_factor_mode_t` {
   
  `kPdbMultFactorAs1` = 0U,  
  `kPdbMultFactorAs10` = 1U,  
  `kPdbMultFactorAs20` = 2U,  
  `kPdbMultFactorAs40` = 3U }

*Defines the type of the multiplication source mode for PDB.*

## Functions

- void `PDB_HAL_Init` (uint32\_t baseAddr)  
*Resets the PDB registers to a known state.*
- static void `PDB_HAL_SetLoadMode` (uint32\_t baseAddr, `pdb_load_mode_t` mode)  
*Sets the load mode for timing registers.*
- static void `PDB_HAL_SetSeqErrIntCmd` (uint32\_t baseAddr, bool enabled)  
*Switches to enable the PDB sequence error interrupt.*
- static void `PDB_HAL_SetSoftTriggerCmd` (uint32\_t baseAddr)  
*Triggers the DAC by software if enabled.*
- static void `PDB_HAL_SetDmaCmd` (uint32\_t baseAddr, bool enable)  
*Switches to enable the PDB DMA support.*
- static void `PDB_HAL_SetPreDivMode` (uint32\_t baseAddr, `pdb_clk_prescaler_div_mode_t` mode)  
*Sets the prescaler divider from the peripheral bus clock for the PDB.*
- static void `PDB_HAL_SetTriggerSrcMode` (uint32\_t baseAddr, `pdb_trigger_src_mode_t` mode)  
*Sets the trigger source mode for the PDB module.*
- static void `PDB_HAL_Enable` (uint32\_t baseAddr)  
*Switches on to enable the PDB module.*
- static void `PDB_HAL_Disable` (uint32\_t baseAddr)  
*Switches off to enable the PDB module.*
- static bool `PDB_HAL_GetIntFlag` (uint32\_t baseAddr)  
*Gets the PDB delay interrupt flag.*
- static void `PDB_HAL_ClearIntFlag` (uint32\_t baseAddr)  
*Clears the PDB delay interrupt flag.*
- static void `PDB_HAL_SetIntCmd` (uint32\_t baseAddr, bool enable)  
*Switches to enable the PDB interrupt.*
- static void `PDB_HAL_SetPreMultFactorMode` (uint32\_t baseAddr, `pdb_mult_factor_mode_t` mode)  
*Sets the PDB prescaler multiplication factor.*
- static void `PDB_HAL_SetContinuousModeCmd` (uint32\_t baseAddr, bool enable)  
*Switches to enable the PDB continuous mode.*
- static void `PDB_HAL_SetLoadRegsCmd` (uint32\_t baseAddr)  
*Loads the delay registers value for the PDB module.*
- static void `PDB_HAL_SetModulusValue` (uint32\_t baseAddr, uint32\_t value)  
*Sets the modulus value for the PDB module.*
- static uint32\_t `PDB_HAL_GetModulusValue` (uint32\_t baseAddr)  
*Gets the modulus value for the PDB module.*
- static uint32\_t `PDB_HAL_GetCounterValue` (uint32\_t baseAddr)  
*Gets the PDB counter value.*
- static void `PDB_HAL_SetIntDelayValue` (uint32\_t baseAddr, uint32\_t value)  
*Sets the interrupt delay milestone of the PDB counter.*
- static uint32\_t `PDB_HAL_GetIntDelayValue` (uint32\_t baseAddr)

## PDB HAL Driver

- Gets the current interrupt delay milestone of the PDB counter.
- void [PDB\\_HAL\\_SetPreTriggerBackToBackCmd](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn, bool enable)  
    Switches to enable the pre-trigger back-to-back mode.
- void [PDB\\_HAL\\_SetPreTriggerOutputCmd](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn, bool enable)  
    Switches to enable the pre-trigger output.
- void [PDB\\_HAL\\_SetPreTriggerCmd](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn, bool enable)  
    Switches to enable the pre-trigger.
- static bool [PDB\\_HAL\\_GetPreTriggerFlag](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn)  
    Gets the flag which indicates whether the PDB counter has reached the pre-trigger delay value.
- void [PDB\\_HAL\\_ClearPreTriggerFlag](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn)  
    Clears the flag which indicates that the PDB counter has reached the pre-trigger delay value.
- static bool [PDB\\_HAL\\_GetPreTriggerSeqErrFlag](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn)  
    Gets the flag which indicates whether a sequence error is detected.
- void [PDB\\_HAL\\_ClearPreTriggerSeqErrFlag](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn)  
    Clears the flag which indicates that a sequence error has been detected.
- void [PDB\\_HAL\\_SetPreTriggerDelayCount](#) (uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn, uint32\_t value)  
    Sets the pre-trigger delay value.
- static void [PDB\\_HAL\\_SetDacExtTriggerInputCmd](#) (uint32\_t baseAddr, uint32\_t dacChn, bool enable)  
    Switches to enable the DAC external trigger input.
- static void [PDB\\_HAL\\_SetDacIntervalTriggerCmd](#) (uint32\_t baseAddr, uint32\_t dacChn, bool enable)  
    Switches to enable the DAC external trigger input.
- static void [PDB\\_HAL\\_SetDacIntervalValue](#) (uint32\_t baseAddr, uint32\_t dacChn, uint32\_t value)  
    Sets the interval value for the DAC trigger.
- static uint32\_t [PDB\\_HAL\\_GetDacIntervalValue](#) (uint32\_t baseAddr, uint32\_t dacChn)  
    Gets the interval value for the DAC trigger.
- void [PDB\\_HAL\\_SetPulseOutCmd](#) (uint32\_t baseAddr, uint32\_t pulseChn, bool enable)  
    Switches to enable the pulse-out trigger.
- static void [PDB\\_HAL\\_SetPulseOutDelayForHigh](#) (uint32\_t baseAddr, uint32\_t pulseChn, uint32\_t value)  
    Sets the counter delay value for the pulse-out goes high.
- static void [PDB\\_HAL\\_SetPulseOutDelayForLow](#) (uint32\_t baseAddr, uint32\_t pulseChn, uint32\_t value)  
    Sets the counter delay value for the pulse-out goes low.

### 17.1.0.10 PDB HAL driver

#### Overview

The PDB HAL driver is used to mask provide a comprehensible way to use PDB hardware.

## 17.1.1 Enumeration Type Documentation

### 17.1.1.1 enum pdb\_status\_t

Enumerator

*kStatus\_PDB\_Success* Success.

*kStatus\_PDB\_InvalidArgument* Invalid argument existed.

*kStatus\_PDB\_Failed* Execution failed.

### 17.1.1.2 enum pdb\_load\_mode\_t

Some timing related registers, such as the MOD, IDLY, CHnDLYm, INTx and POyDLY, buffer the setting values. Only the load operation is triggered. The setting value is loaded from a buffer and takes effect. There are four loading modes to fit different applications.

Enumerator

*kPdbLoadImmediately* Loaded immediately after load operation.

*kPdbLoadAtModuloCounter* Loaded when counter hits the modulo after load operation.

*kPdbLoadAtNextTrigger* Loaded when detecting an input trigger after load operation.

*kPdbLoadAtModuloCounterOrNextTrigger* Loaded when counter hits the modulo or detecting an input trigger after load operation.

### 17.1.1.3 enum pdb\_clk\_prescaler\_div\_mode\_t

Enumerator

*kPdbClkPreDivBy1* Counting divided by multiplication factor selected by MULT.

*kPdbClkPreDivBy2* Counting divided by multiplication factor selected by 2 times ofMULT.

*kPdbClkPreDivBy4* Counting divided by multiplication factor selected by 4 times ofMULT.

*kPdbClkPreDivBy8* Counting divided by multiplication factor selected by 8 times ofMULT.

*kPdbClkPreDivBy16* Counting divided by multiplication factor selected by 16 times ofMULT.

*kPdbClkPreDivBy32* Counting divided by multiplication factor selected by 32 times ofMULT.

*kPdbClkPreDivBy64* Counting divided by multiplication factor selected by 64 times ofMULT.

*kPdbClkPreDivBy128* Counting divided by multiplication factor selected by 128 times ofMULT.

### 17.1.1.4 enum pdb\_trigger\_src\_mode\_t

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger.

## PDB HAL Driver

Enumerator

- kPdbTrigger0*** Select trigger-In 0.
- kPdbTrigger1*** Select trigger-In 1.
- kPdbTrigger2*** Select trigger-In 2.
- kPdbTrigger3*** Select trigger-In 3.
- kPdbTrigger4*** Select trigger-In 4.
- kPdbTrigger5*** Select trigger-In 5.
- kPdbTrigger6*** Select trigger-In 6.
- kPdbTrigger7*** Select trigger-In 7.
- kPdbTrigger8*** Select trigger-In 8.
- kPdbTrigger9*** Select trigger-In 8.
- kPdbTrigger10*** Select trigger-In 10.
- kPdbTrigger11*** Select trigger-In 11.
- kPdbTrigger12*** Select trigger-In 12.
- kPdbTrigger13*** Select trigger-In 13.
- kPdbTrigger14*** Select trigger-In 14.
- kPdbSoftTrigger*** Select software trigger.

### 17.1.1.5 enum pdb\_mult\_factor\_mode\_t

Selects the multiplication factor of the prescaler divider for the PDB counter clock.

Enumerator

- kPdbMultFactorAs1*** Multiplication factor is 1.
- kPdbMultFactorAs10*** Multiplication factor is 10.
- kPdbMultFactorAs20*** Multiplication factor is 20.
- kPdbMultFactorAs40*** Multiplication factor is 40.

## 17.1.2 Function Documentation

### 17.1.2.1 void PDB\_HAL\_Init ( uint32\_t baseAddr )

This function resets the PDB registers to a known state. This state is defined in a reference manual and is power on reset value.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

#### 17.1.2.2 static void PDB\_HAL\_SetLoadMode ( uint32\_t *baseAddr*, pdb\_load\_mode\_t *mode* ) [inline], [static]

This function sets the load mode for some timing registers including MOD, IDLY, CHnDLYm, INTx and POyDLY.

Parameters

|                 |                                              |
|-----------------|----------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.        |
| <i>mode</i>     | Selection of mode, see to "pdb_load_mode_t". |

#### 17.1.2.3 static void PDB\_HAL\_SetSeqErrIntCmd ( uint32\_t *baseAddr*, bool *enabled* ) [inline], [static]

This function switches to enable the PDB sequence error interrupt.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>enable</i>   | The switcher to assert the feature.   |

#### 17.1.2.4 static void PDB\_HAL\_SetSoftTriggerCmd ( uint32\_t *baseAddr* ) [inline], [static]

If enabled, this function triggers the DAC by using software.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

#### 17.1.2.5 static void PDB\_HAL\_SetDmaCmd ( uint32\_t *baseAddr*, bool *enable* ) [inline], [static]

This function switches to enable the PDB DMA support.

## PDB HAL Driver

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>enable</i>   | The switcher to assert the feature.   |

### 17.1.2.6 static void PDB\_HAL\_SetPreDivMode ( uint32\_t *baseAddr*, pdb\_clk\_prescaler\_div\_mode\_t *mode* ) [inline], [static]

This function sets the prescaler divider from the peripheral bus clock for the PDB.

Parameters

|                 |                                                           |
|-----------------|-----------------------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.                     |
| <i>mode</i>     | Selection of mode, see to "pdb_clk_prescaler_div_mode_t". |

### 17.1.2.7 static void PDB\_HAL\_SetTriggerSrcMode ( uint32\_t *baseAddr*, pdb\_trigger\_src\_mode\_t *mode* ) [inline], [static]

This function sets the trigger source mode for the PDB module.

Parameters

|                 |                                                     |
|-----------------|-----------------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.               |
| <i>mode</i>     | Selection of mode, see to "pdb_trigger_src_mode_t". |

### 17.1.2.8 static void PDB\_HAL\_Enable ( uint32\_t *baseAddr* ) [inline], [static]

This function switches on to enable the PDB module.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

### 17.1.2.9 static void PDB\_HAL\_Disable ( uint32\_t *baseAddr* ) [inline], [static]

This function switches off to enable the PDB module.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

#### 17.1.2.10 static bool PDB\_HAL\_GetIntFlag ( uint32\_t *baseAddr* ) [inline], [static]

This function gets the PDB delay interrupt flag.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

Returns

Flat status, true if the flag is set.

#### 17.1.2.11 static void PDB\_HAL\_ClearIntFlag ( uint32\_t *baseAddr* ) [inline], [static]

This function clears PDB delay interrupt flag.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

Returns

Flat status, true if the flag is set.

#### 17.1.2.12 static void PDB\_HAL\_SetIntCmd ( uint32\_t *baseAddr*, bool *enable* ) [inline], [static]

This function switches to enable the PDB interrupt.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>enable</i>   | The switcher to assert the feature.   |

### 17.1.2.13 static void PDB\_HAL\_SetPreMultFactorMode ( *uint32\_t baseAddr*,                   *pdb\_mult\_factor\_mode\_t mode* ) [inline], [static]

This function sets the PDB prescaler multiplication factor.

Parameters

|                 |                                                     |
|-----------------|-----------------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.               |
| <i>mode</i>     | Selection of mode, see to "pdb_mult_factor_mode_t". |

#### 17.1.2.14 static void PDB\_HAL\_SetContinuousModeCmd ( uint32\_t *baseAddr*, bool *enable* ) [inline], [static]

This function switches to enable the PDB continuous mode.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>enable</i>   | The switcher to assert the feature.   |

#### 17.1.2.15 static void PDB\_HAL\_SetLoadRegsCmd ( uint32\_t *baseAddr* ) [inline], [static]

This function sets the LDOKE bit and loads the delay registers value. Writing one to this bit updates the internal registers MOD, IDLY, CHnDLYm, DACINTx, and POyDLY with the values written to their buffers. The MOD, IDLY, CHnDLYm, DACINTx, and POyDLY take effect according to the load mode settings.

After one is written to the LDOKE bit, the values in the buffers of above mentioned registers are not effective and cannot be written until the values in the buffers are loaded into their internal registers. The LDOKE can be written only when the the PDB is enabled or as alone with it. It is automatically cleared either when the values in the buffers are loaded into the internal registers or when the PDB is disabled.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

#### 17.1.2.16 static void PDB\_HAL\_SetModulusValue ( uint32\_t *baseAddr*, uint32\_t *value* ) [inline], [static]

This function sets the modulus value for the PDB module. When the counter reaches the setting value, it is automatically reset to zero. When in continuous mode, the counter begins to increase again.

## PDB HAL Driver

Parameters

|                 |                                                   |
|-----------------|---------------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.             |
| <i>value</i>    | The setting value of upper limit for PDB counter. |

### 17.1.2.17 static uint32\_t PDB\_HAL\_GetModulusValue ( uint32\_t *baseAddr* ) [inline], [static]

This function gets the modulus value for the PDB module.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

Returns

The current value of upper limit for counter.

### 17.1.2.18 static uint32\_t PDB\_HAL\_GetCounterValue ( uint32\_t *baseAddr* ) [inline], [static]

This function gets the PDB counter value.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

Returns

The current counter value.

### 17.1.2.19 static void PDB\_HAL\_SetIntDelayValue ( uint32\_t *baseAddr*, uint32\_t *value* ) [inline], [static]

This function sets the interrupt delay milestone of the PDB counter. If enabled, a PDB interrupt is generated when the counter is equal to the setting value.

Parameters

|                 |                                                                 |
|-----------------|-----------------------------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.                           |
| <i>value</i>    | The setting value for interrupt delay milestone of PDB counter. |

#### 17.1.2.20 static uint32\_t PDB\_HAL\_GetIntDelayValue ( uint32\_t *baseAddr* ) [inline], [static]

This function gets the current interrupt delay milestone of the PDB counter.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
|-----------------|---------------------------------------|

Returns

The current setting value for interrupt delay milestone of PDB counter.

#### 17.1.2.21 void PDB\_HAL\_SetPreTriggerBackToBackCmd ( uint32\_t *baseAddr*, uint32\_t *chn*, uint32\_t *preChn*, bool *enable* )

This function switches to enable the pre-trigger back-to-back mode.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |
| <i>enable</i>   | Switcher to assert the feature.       |

#### 17.1.2.22 void PDB\_HAL\_SetPreTriggerOutputCmd ( uint32\_t *baseAddr*, uint32\_t *chn*, uint32\_t *preChn*, bool *enable* )

This function switches to enable pre-trigger output.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |
| <i>enable</i>   | Switcher to assert the feature.       |

### 17.1.2.23 void PDB\_HAL\_SetPreTriggerCmd ( *uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn, bool enable* )

This function switches to enable the pre-trigger.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |
| <i>enable</i>   | Switcher to assert the feature.       |

### 17.1.2.24 static bool PDB\_HAL\_GetPreTriggerFlag ( *uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn* ) [inline], [static]

This function gets the flag which indicates the PDB counter has reached the pre-trigger delay value.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |

Returns

Flag status. True if the event is asserted.

### 17.1.2.25 void PDB\_HAL\_ClearPreTriggerFlag ( *uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn* )

This function clears the flag which indicates that the PDB counter has reached the pre-trigger delay value.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |

**17.1.2.26 static bool PDB\_HAL\_GetPreTriggerSeqErrFlag ( *uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn* ) [inline], [static]**

This function gets the flag which indicates whether a sequence error is detected.

## PDB HAL Driver

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |

Returns

Flag status. True if the event is asserted.

### 17.1.2.27 void PDB\_HAL\_ClearPreTriggerSeqErrFlag ( *uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn* )

This function clears the flag which indicates that the sequence error has been detected.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>chn</i>      | ADC instance index for trigger.       |
| <i>preChn</i>   | ADC channel group index for trigger.  |

### 17.1.2.28 void PDB\_HAL\_SetPreTriggerDelayCount ( *uint32\_t baseAddr, uint32\_t chn, uint32\_t preChn, uint32\_t value* )

This function sets the pre-trigger delay value.

Parameters

|                 |                                              |
|-----------------|----------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.        |
| <i>chn</i>      | ADC instance index for trigger.              |
| <i>preChn</i>   | ADC channel group index for trigger.         |
| <i>value</i>    | Setting value for pre-trigger's delay value. |

### 17.1.2.29 static void PDB\_HAL\_SetDacExtTriggerInputCmd ( *uint32\_t baseAddr, uint32\_t dacChn, bool enable* ) [inline], [static]

This function switches to enable the DAC external trigger input.

Parameters

|                 |                                              |
|-----------------|----------------------------------------------|
| <i>baseAddr</i> | Register base address for the module.        |
| <i>dacChn</i>   | DAC instance index for trigger.              |
| <i>value</i>    | Setting value for pre-trigger's delay value. |

#### 17.1.2.30 static void PDB\_HAL\_SetDacIntervalTriggerCmd ( *uint32\_t baseAddr, uint32\_t dacChn, bool enable* ) [inline], [static]

This function switches to enable the DAC external trigger input.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>dacChn</i>   | DAC instance index for trigger.       |
| <i>enable</i>   | Switcher to assert the feature.       |

#### 17.1.2.31 static void PDB\_HAL\_SetDacIntervalValue ( *uint32\_t baseAddr, uint32\_t dacChn, uint32\_t value* ) [inline], [static]

This function sets the interval value for the DAC trigger.

Parameters

|                 |                                         |
|-----------------|-----------------------------------------|
| <i>baseAddr</i> | Register base address for the module.   |
| <i>dacChn</i>   | DAC instance index for trigger.         |
| <i>value</i>    | Setting value for DAC trigger interval. |

#### 17.1.2.32 static uint32\_t PDB\_HAL\_GetDacIntervalValue ( *uint32\_t baseAddr, uint32\_t dacChn* ) [inline], [static]

This function gets the interval value for the DAC trigger.

Parameters

## PDB HAL Driver

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>dacChn</i>   | DAC instance index for trigger.       |

Returns

The current setting value for DAC trigger interval.

### **17.1.2.33 void PDB\_HAL\_SetPulseOutCmd ( uint32\_t *baseAddr*, uint32\_t *pulseChn*, bool *enable* )**

This function switches to enable the pulse-out trigger.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>pulseChn</i> | Pulse-out channel index for trigger.  |
| <i>enable</i>   | Switcher to assert the feature.       |

### **17.1.2.34 static void PDB\_HAL\_SetPulseOutDelayForHigh ( uint32\_t *baseAddr*, uint32\_t *pulseChn*, uint32\_t *value* ) [inline], [static]**

This function sets the counter delay value for the pulse-out goes high.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>pulseChn</i> | Pulse-out channel index for trigger.  |
| <i>value</i>    | Setting value for PDB delay .         |

### **17.1.2.35 static void PDB\_HAL\_SetPulseOutDelayForLow ( uint32\_t *baseAddr*, uint32\_t *pulseChn*, uint32\_t *value* ) [inline], [static]**

This function sets the counter delay value for the pulse-out goes low.

Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>baseAddr</i> | Register base address for the module. |
| <i>pulseChn</i> | Pulse-out channel index for trigger.  |
| <i>value</i>    | Setting value for PDB delay .         |

### 17.2 PDB Peripheral Driver

This chapter describes the programming interface of the PDB Peripheral driver.

#### Data Structures

- struct [pdb\\_user\\_config\\_t](#)  
*Defines the structure to configure the PDB counter. [More...](#)*
- struct [pdb\\_adc\\_pre\\_trigger\\_config\\_t](#)  
*Defines the structure to configure the ADC pre-trigger in the PDB module. [More...](#)*
- struct [pdb\\_dac\\_trigger\\_config\\_t](#)  
*Defines the structure to configuring the DAC trigger in the PDB module. [More...](#)*
- struct [pdb\\_pulse\\_out\\_trigger\\_config\\_t](#)  
*Defines the structure to configure the pulse-out trigger in the PDB module. [More...](#)*
- struct [pdb\\_state\\_t](#)  
*Internal driver state information. [More...](#)*

#### Typedefs

- typedef void(\* [pdb\\_callback\\_t](#) )(void)  
*Defines the type of user-defined callback function for the PDB module.*

#### Enumerations

- enum [pdb\\_adc\\_pre\\_trigger\\_flag\\_t](#) {  
  kPdbAdcPreChnFlag = 0U,  
  kPdbAdcPreChnErrFlag = 1U }  
*Defines the type of flag for PDB pre-trigger events.*

#### Functions

- [pdb\\_status\\_t PDB\\_DRV\\_StructInitUserConfigForSoftTrigger](#) ([pdb\\_user\\_config\\_t](#) \*userConfigPtr)  
*Fills the initial user configuration for software trigger mode.*
- [pdb\\_status\\_t PDB\\_DRV\\_Init](#) (uint32\_t instance, [pdb\\_user\\_config\\_t](#) \*userConfigPtr, [pdb\\_state\\_t](#) \*userStatePtr)  
*Initializes the PDB counter and trigger input.*
- void [PDB\\_DRV\\_Deinit](#) (uint32\_t instance)  
*De-initializes the PDB module.*
- void [PDB\\_DRV\\_SoftTriggerCmd](#) (uint32\_t instance)  
*Triggers the PDB with a software trigger.*
- uint32\_t [PDB\\_DRV\\_GetCurrentCounter](#) (uint32\_t instance)  
*Gets the current counter value in the PDB module.*
- bool [PDB\\_DRV\\_GetPdbCounterIntFlag](#) (uint32\_t instance)  
*Gets the PDB interrupt flag.*
- void [PDB\\_DRV\\_ClearPdbCounterIntFlag](#) (uint32\_t instance)  
*Clears the interrupt flag.*

- `pdb_status_t PDB_DRV_EnableAdcPreTrigger` (uint32\_t instance, uint32\_t adcChn, uint32\_t preChn, `pdb_adc_pre_trigger_config_t` \*adcPreTriggerConfigPtr)
 

*Enables the ADC pre-trigger with its configuration.*
- `void PDB_DRV_DisableAdcPreTrigger` (uint32\_t instance, uint32\_t adcChn, uint32\_t preChn)
 

*Disables the ADC pre-trigger.*
- `bool PDB_DRV_GetAdcPreTriggerFlag` (uint32\_t instance, uint32\_t adcChn, uint32\_t preChn, `pdb_adc_pre_trigger_flag_t` flag)
 

*Gets the ADC pre-trigger flag.*
- `void PDB_DRV_ClearAdcPreTriggerFlag` (uint32\_t instance, uint32\_t adcChn, uint32\_t preChn, `pdb_adc_pre_trigger_flag_t` flag)
 

*Clears the ADC pre-trigger flag.*
- `pdb_status_t PDB_DRV_EnableDacTrigger` (uint32\_t instance, uint32\_t dacChn, `pdb_dac_trigger_config_t` \*dacTriggerConfigPtr)
 

*Enables the DAC trigger with its configuration.*
- `void PDB_DRV_DisableDacTrigger` (uint32\_t instance, uint32\_t dacChn)
 

*Disables the DAC trigger.*
- `pdb_status_t PDB_DRV_EnablePulseOutTrigger` (uint32\_t instance, uint32\_t pulseChn, `pdb_pulse_out_trigger_config_t` \*pulseOutTriggerConfigPtr)
 

*Enables the pulse-out trigger with its configuration.*
- `void PDB_DRV_DisablePulseOutTrigger` (uint32\_t instance, uint32\_t pulseChn)
 

*Disables the pulse-out trigger.*
- `pdb_status_t PDB_DRV_InstallCallback` (uint32\_t instance, `pdb_callback_t` userCallback)
 

*Installs the user-defined callback for PDB module.*
- `void PDB_DRV_IRQHandler` (uint32\_t instance)
 

*Driver-defined ISR in the PDB module.*

### 17.2.0.36 PDB Peripheral Driver

#### Overview

The PDB peripheral driver configures the PDB (Programmable Delay Block). It handles the triggers for ADC and DAC and pulse out to the CMP and the PDB counter itself.

#### Driver model building

There is one main PDB counter for all triggers. When the indicated external trigger input arrives, the PDB counter launches to be increased by setting clock. There are also counter trigger milestones, for ADC and DAC, and pulse out to the CMP and the PDB counter itself, waiting for the PDB counter. Once the PDB counter hits each milestone, the critical delay value, the corresponding event is triggered and the trigger signal sends out to trigger other peripherals. Therefore, the PDB module is a collector and manager of triggers.

### Initialization

The core feature of the PDB module is a programmable timer/counter. Additional features enable and set the milestone for the corresponding trigger. Therefore, the PDB module is first initialized as a programmable timer. To initialize the PDB driver, a configuration structure with type of "pdb\_user\_config\_t" is needed and should be filled with an available configuration. The API of the [PDB\\_DRV\\_StructInitUserConfigForSoftTrigger\(\)](#) function can provide an acceptable configuration to make the PDB work. But it is almost not fitting the your own case. So please provide the real configuration according to application. Call the API of [PDB\\_DRV\\_Init\(\)](#) function to initialize the PDB timer/counter.

All the triggers share the same counter. However, the DAC trigger does not share the same counting circle with other triggers. When the DAC's internal circle ends the trigger is generated and the internal circle restarts from the beginning.

For timing setting

The basic timing/counting step is set when initializing the main PDB counter:

The basic timing/counting step =  $F_{BusClkHHz} / \text{pdb\_user\_config\_t.clkPrescalerDivMode} / \text{pdb\_user\_config\_t.multFactorMode}$

Just as the  $F_{BusClkHHz}$  is the frequency of bus clock in Hz, "clkPrescalerDivMode" and "multFactorMode" are in the [pdb\\_user\\_config\\_t](#) structure. All the triggers milestones are based on this step unit.

### Call diagram

Four kinds of typical use cases are designed for the PDB module.

- Normal Timer/Counter. Normal Timer/Counter is the basic case of using PDB. The Timer/Counter starts after the PDB is initialized. The milestone for PDB Timer/Counter is set. After it is triggered, when the counter hits the milestone, it causes the interrupt request if enabled. In continuous mode, when the counter hits the upper limitation, it returns zero and counts again.
- Trigger for ADC module. When the ADC trigger is enabled, a delay value for ADC trigger is set as the milestone. At least two ADC channel groups are provided. Likewise, there are more than two pre-triggers for ADC. Each pre-trigger is related to one channel group and can be enabled separately in the PDB module. When the PDB counter hits the milestone for the ADC pre-trigger, it triggers the ADC's conversion on the indicated channel group. To maximize the feature, the ADC should be configured to enable the hardware trigger mode.
- Trigger for the DAC module. A standalone DAC counter exists in the PDB module to trigger the DAC module. The user can set the upper limitation for the DAC counter. Once the counter reaches the upper limitation, it turns the DAC counter to zero and count again. When the DAC counter hits the upper limitation, an DAC trigger is generated to trigger the DAC. This trigger updates the pointer of the DAC buffer. Although the DAC counter has its own setting for the upper limitation, it shares the same input trigger source, clock source and reset control logic with the main PDB counter. When the PDB counter resets to zero, it forces the DAC counter to reset to zero. To maximize this feature, the DAC should be configured to enable the DAC buffer and hardware trigger mode.
- Trigger for pulse out to the CMP module. The pulse-out trigger is attached to the main PDB counter.

There are two milestones for each pulse out channel. One milestone is for level high and the other milestone is for level low, which makes a sample window for the CMP module.

These are the examples to initialize and configure the PDB driver for typical use cases.

Normal Timer/Counter:

```
/* pdb_test_normal_timer.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_pdb_driver.h"
#include "fsl_os_abstraction.h"

#define PDB_TEST_TIMER_ROUNT_TIMES (10U)

static volatile uint32_t MyPdbIntCounter = 0U;
static pdb_state_t MyPdbStateStruct;

extern void MyNop(void);
static void PDB_ISR_Counter(void);

void PDB_TEST_NormalTimer(uint32_t instance)
{
 pdb_user_config_t MyPdbUserConfigStruct;

 /* Prepare the configuration structure. */
 PDB_DRV_StructInitUserConfigForSoftTrigger(&
 MyPdbUserConfigStruct);
 MyPdbUserConfigStruct.continuousModeEnable = true;
 MyPdbUserConfigStruct.delayValue = 0x002FU;
 MyPdbUserConfigStruct.pdbModulusValue = 0x004FU;

 /* Initialize PDB counter. */
 PDB_DRV_Init(instance, &MyPdbUserConfigStruct, &MyPdbStateStruct);

 /* Install the callback function. */
 PDB_DRV_InstallCallback(instance, PDB_ISR_Counter);

 /* Trigger the PDB. */
 PDB_DRV_SoftTriggerCmd(instance);

 /* Wait for the counter. */
 while (1U)
 {
 if (MyPdbIntCounter >= PDB_TEST_TIMER_ROUNT_TIMES)
 {
 printf("PDB counter goes %d rounds.\r\n", MyPdbIntCounter);
 break;
 }
 MyNop();
 }

 /* De-initialize the PDB. */
 PDB_DRV_Deinit(instance);

 MyPdbIntCounter = 0U;
}

static void PDB_ISR_Counter(void)
{
 MyPdbIntCounter++;
}
```

Trigger for ADC module:

## PDB Peripheral Driver

```
/* pdb_test_adc_trigger.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_pdb_driver.h"
#include "fsl_adc_driver.h"
#include "fsl_os_abstraction.h"

#define PDB_TEST_ADC_TRIGGER_INSTANCE (0U)
#define PDB_TEST_ADC_CHANNEL_GROUP (1U)
#define PDB_TEST_ADC_CHANNEL_ID (26U)
#define PDB_TEST_ADC_TRIGGER_TIMES (10U)

static adc_state_t MyAdcState;
static pdb_state_t MyPdbStateStruct;
static volatile uint32_t MyAdcIntCounter = 0U;
static volatile uint32_t MyPdbIntCounter = 0U;

static void PDB_TEST_InitAdc(uint32_t instance, uint32_t chnGroup, uint32_t chn);
static void ADC_TEST_MyIsr(void);
static void PDB_ISR_Counter(void);
extern void MyNop(void);

void PDB_TEST_AdcTrigger(uint32_t instance)
{
 pdb_user_config_t MyPdbUserConfigStruct;
 pdb_adc_pre_trigger_config_t MyPdbAdcTriggerConfigStruct;
 uint32_t i;

 /* Initialize the ADC that would be triggered. */
 PDB_TEST_InitAdc(PDB_TEST_ADC_TRIGGER_INSTANCE, PDB_TEST_ADC_CHANNEL_GROUP, PDB_TEST_ADC_CHANNEL_ID);

 /* Prepare the configuration structure. */
 PDB_DRV_StructInitUserConfigForSoftTrigger(&
 MyPdbUserConfigStruct);
 MyPdbUserConfigStruct.continuousModeEnable = true;
 MyPdbUserConfigStruct.delayValue = 0x002FU;
 MyPdbUserConfigStruct.pdbModulusValue = 0x004FU;

 /* Initialize PDB counter. */
 PDB_DRV_Init(instance, &MyPdbUserConfigStruct, &MyPdbStateStruct);

 /* Install the callback function. */
 PDB_DRV_InstallCallback(instance, PDB_ISR_Counter);

 /* Initialize the ADC trigger in PDB. */
 MyPdbAdcTriggerConfigStruct.backToBackModeEnable = false;
 MyPdbAdcTriggerConfigStruct.triggerOutEnable = true;
 MyPdbAdcTriggerConfigStruct.delayValue = 0x10U;
 PDB_DRV_EnableAdcPreTrigger(instance, PDB_TEST_ADC_TRIGGER_INSTANCE,
 PDB_TEST_ADC_CHANNEL_GROUP, &MyPdbAdcTriggerConfigStruct);

 /* Trigger the PDB. */
 PDB_DRV_SoftTriggerCmd(instance);

 while (1U)
 {
 if (MyAdcIntCounter >= PDB_TEST_ADC_TRIGGER_TIMES)
 {
 printf("ADC has been triggered by %d times.\r\n",
 MyAdcIntCounter);
 break;
 }

 MyNop();
 }
}
```

```

PDB_DRV_DisableAdcPreTrigger(instance, PDB_TEST_ADC_TRIGGER_INSTANCE,
 PDB_TEST_ADC_CHANNEL_GROUP);
PDB_DRV_Deinit(instance);
ADC_DRV_Deinit(PDB_TEST_ADC_TRIGGER_INSTANCE);

MyAdcIntCounter = 0U;
}

static void PDB_TEST_InitAdc(uint32_t instance, uint32_t chnGroup, uint32_t chn)
{
 adc_calibration_param_t MyAdcCalibraitionParam;
 adc_user_config_t MyAdcUserConfig;
 adc_chn_config_t MyChnConfig;

 /* Auto calibraion. */
 ADC_DRV_GetAutoCalibrationParam(instance, &MyAdcCalibraitionParam);
 ADC_DRV_SetCalibrationParam(instance, &MyAdcCalibraitionParam);

 /* Initialization for interrupt mode but not continuous mode. */
 ADC_DRV_StructInitUserConfigForIntMode(&MyAdcUserConfig);
 MyAdcUserConfig.continuousConvEnable = false; /* One conversion per-trigger. */
 MyAdcUserConfig.hwTriggerEnable = true; /* Hardware trigger. */
 ADC_DRV_Init(instance, &MyAdcUserConfig, &MyAdcState);

 /* Install Callback function into ISR. */
 ADC_DRV_InstallCallback(instance, chnGroup, ADC_TEST_MyIsr);

 /* Set the channel. */
 MyChnConfig.chnNum = chn;
 MyChnConfig.diffEnable = false;
 MyChnConfig.intEnable = true;
 MyChnConfig.chnMux = kAdcChnMuxOfDefault;
 ADC_DRV_ConfigConvChn(instance, chnGroup, &MyChnConfig);
}

static void ADC_TEST_MyIsr(void)
{
 MyAdcIntCounter++;
}

static void PDB_ISR_Counter(void)
{
 MyPdbIntCounter++;
}

```

Trigger for DAC module:

```

/* pdb_test_dac_trigger.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <fsl_pdb_driver.h>
#include <fsl_dac_driver.h>
#include "fsl_os_abstraction.h"

#define PDB_TEST_TRIGGER_DAC_INSTANCE (0U)
#define PDB_TEST_DAC_ARR_LEN (16U)
#define PDB_TEST_DAC_TRIGGER_TIMES (10U)

uint16_t MyPdbDemoDacBuff[PDB_TEST_DAC_ARR_LEN] =
{
 0U, 256U, 512U, 768U,
 1024U, 1280U, 1536U, 1792U,
 2048U, 2304U, 2560U, 2816U,
 3072U, 3328U, 3584U, 3840U

```

## PDB Peripheral Driver

```
};

static dac_state_t MyDacStateStructForBufferFIFO;
static pdb_state_t MyPdbStateStruct;
static volatile uint32_t MyPdbIntCounter = 0U;
static volatile uint32_t MyDacIntCounter = 0U;

extern void MyNop(void);
static void DAC_ISR_Buffer(void);
static void PDB_TEST_InitDac(uint32_t instance);
static void PDB_ISR_Counter(void);

void PDB_TEST_DacTrigger(uint32_t instance)
{
 pdb_user_config_t MyPdbUserConfigStruct;
 pdb_dac_trigger_config_t MyPdbDacTriggerConfigStruct;
 uint32_t i;

 /* Initialize the DAC module. */
 PDB_TEST_InitDac(PDB_TEST_TRIGGER_DAC_INSTANCE);

 /* Prepare the configuration structure. */
 PDB_DRV_StructInitUserConfigForSoftTrigger(&
 MyPdbUserConfigStruct);
 MyPdbUserConfigStruct.continuousModeEnable = true;
 MyPdbUserConfigStruct.delayValue = 0x002FU;
 MyPdbUserConfigStruct.pdbModulusValue = 0x004FU;

 /* Initialize PDB counter. */
 PDB_DRV_Init(instance, &MyPdbUserConfigStruct, &MyPdbStateStruct);

 /* Install the PDB ISR. */
 PDB_DRV_InstallCallback(instance, PDB_ISR_Counter);

 /* Initialize the DAC trigger. */
 MyPdbDacTriggerConfigStruct.extTriggerInputEnable = false;
 MyPdbDacTriggerConfigStruct.intervalValue = 4U;
 PDB_DRV_EnableDacTrigger(instance, PDB_TEST_TRIGGER_DAC_INSTANCE,
 &MyPdbDacTriggerConfigStruct);

 /* Trigger the PDB. */
 PDB_DRV_SoftTriggerCmd(instance);

 while (1)
 {
 if (MyPdbIntCounter >= PDB_TEST_DAC_TRIGGER_TIMES)
 {
 printf("DAC buffer interrupt Start/Upper/Watermark counter: %d.\r\n",
 MyDacIntCounter);
 break;
 }

 MyNop();
 }

 PDB_DRV_DisableDacTrigger(instance, PDB_TEST_TRIGGER_DAC_INSTANCE);
 PDB_DRV_Deinit(instance);
 DAC_DRV_Deinit(PDB_TEST_TRIGGER_DAC_INSTANCE);
 MyDacIntCounter = 0U;
 MyPdbIntCounter = 0U;
}

static void PDB_TEST_InitDac(uint32_t instance)
{
 dac_user_config_t MyDacUserConfigStruct;
 dac_buff_config_t MyDacBuffConfigStruct;

 /* Initialize the DAC converter. */
}
```

```

DAC_DRV_StructInitUserConfigNormal(&MyDacUserConfigStruct);
MyDacUserConfigStruct.triggerMode = kDacTriggerByHardware; /* Hardware
trigger. */

DAC_DRV_Init(instance, &MyDacUserConfigStruct);

/* Enable buffer. */
/* Enable the feature of DAC internal buffer. */
MyDacBuffConfigStruct.bufIndexWatermarkIntEnable = true;
MyDacBuffConfigStruct.bufIndexStartIntEnable = true;
MyDacBuffConfigStruct.bufIndexUpperIntEnable = true;
MyDacBuffConfigStruct.dmaEnable = false;
MyDacBuffConfigStruct.watermarkMode = kDacBuffWatermarkFromUpperAs2Word
;
MyDacBuffConfigStruct.bufWorkMode = kDacBuffWorkAsFIFOMode;
MyDacBuffConfigStruct.bufUpperIndex = PDB_TEST_DAC_ARR_LEN - 1;
DAC_DRV_EnableBuff(instance, &MyDacBuffConfigStruct, &MyDacStateStructForBufferFIFO);

/* Fill the buffer with setting data. */
DAC_DRV_SetBuffValue(instance, 0U, PDB_TEST_DAC_ARR_LEN, MyPdbDemoDacBuff);
/* Register the callback function for DAC buffer event. */
DAC_DRV_InstallCallback(instance, DAC_ISR_Buffer);
}

static void DAC_ISR_Buffer(void)
{
 MyDacIntCounter++;
}

static void PDB_ISR_Counter(void)
{
 MyPdbIntCounter++;
}

```

### Trigger for Pulse-out:

```

/* pdb_test_pulse_out_trigger.c */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "fsl_pdb_driver.h"
#include "fsl_os_abstraction.h"

#define PDB_TEST_PULSE_OUT_CHANNEL_ID (0U)
#define PDB_TEST_PULSE_OUT_TRIGGER_TIMES (10U)

static pdb_state_t MyPdbStateStruct;
static volatile uint32_t MyPdbIntCounter = 0U;
extern void MyNop(void);
static void PDB_ISR_Counter(void);

void PDB_TEST_PulseOutTrigger(uint32_t instance)
{
 pdb_user_config_t MyPdbUserConfigStruct;
 pdb_pulse_out_trigger_config_t MyPulseOutTriggerConfigStruct;
 uint32_t i, j;

 /* Prepare the configuration structure. */
 PDB_DRV_StructInitUserConfigForSoftTrigger(&
 MyPdbUserConfigStruct);
 MyPdbUserConfigStruct.continuousModeEnable = true;
 MyPdbUserConfigStruct.delayValue = 0x002FU;
 MyPdbUserConfigStruct.pdbModulusValue = 0x004FU;

 /* Initialize PDB counter. */

```

## PDB Peripheral Driver

```
PDB_DRV_Init(instance, &MyPdbUserConfigStruct, &MyPdbStateStruct);

/* Install the callback function. */
PDB_DRV_InstallCallback(instance, PDB_ISR_Counter);

/* Initialize the Pulse out in PDB. */
MyPulseOutTriggerConfigStruct.pulseOutHighValue = 0x0010U;
MyPulseOutTriggerConfigStruct.pulseOutLowValue = 0x0020U;
PDB_DRV_EnablePulseOutTrigger(instance, PDB_TEST_PULSE_OUT_CHANNEL_ID, &
 MyPulseOutTriggerConfigStruct);

/* Trigger the PDB. */
PDB_DRV_SoftTriggerCmd(instance);

while (1U)
{
 if (MyPdbIntCounter >= PDB_TEST_PULSE_OUT_TRIGGER_TIMES)
 {
 printf("Pulse-out has been triggered by %d times.\r\n",
 MyPdbIntCounter);
 break;
 }
 MyNop();
}

PDB_DRV_DisablePulseOutTrigger(instance, PDB_TEST_PULSE_OUT_CHANNEL_ID);
PDB_DRV_Deinit(instance);
MyPdbIntCounter = 0U;
}

static void PDB_ISR_Counter(void)
{
 MyPdbIntCounter++;
}
```

### 17.2.1 Data Structure Documentation

#### 17.2.1.1 struct pdb\_user\_config\_t

This structure keeps the configuration for the PDB basic counter. The PDB counter is the core part of the PDB module. When initialized, the PDB module can act as a simple counter.

#### Data Fields

- **pdb\_load\_mode\_t loadMode**  
*Selects the mode to load timing registers after set load operation.*
- **bool seqErrIntEnable**  
*Switch to enable the PDB sequence error interrupt.*
- **bool dmaEnable**  
*Switch to enable DMA support for PDB instead of interrupt.*
- **pdb\_clk\_prescaler\_div\_mode\_t clkPrescalerDivMode**  
*Select the prescaler that counting uses the peripheral clock divided by multiplication factor.*
- **pdb\_trigger\_src\_mode\_t triggerSrcMode**  
*Select the input source of trigger.*
- **bool intEnable**  
*Switch to enable the PDB interrupt for counter reaches to the modulus.*

- **pdb\_mult\_factor\_mode\_t multFactorMode**  
*Select the multiplication factor for prescalar.*
- **bool continuousModeEnable**  
*Switch to enable the continuous mode.*
- **uint32\_t pdbModulusValue**  
*Set the value for PDB counter's modulus.*
- **uint32\_t delayValue**  
*Set the value for PDB counter to cause PDB interrupt.*

#### 17.2.1.1.0.40 Field Documentation

17.2.1.1.0.40.1 **pdb\_load\_mode\_t pdb\_user\_config\_t::loadMode**

17.2.1.1.0.40.2 **bool pdb\_user\_config\_t::seqErrIntEnable**

17.2.1.1.0.40.3 **bool pdb\_user\_config\_t::dmaEnable**

17.2.1.1.0.40.4 **pdb\_clk\_prescaler\_div\_mode\_t pdb\_user\_config\_t::clkPrescalerDivMode**

17.2.1.1.0.40.5 **pdb\_trigger\_src\_mode\_t pdb\_user\_config\_t::triggerSrcMode**

17.2.1.1.0.40.6 **bool pdb\_user\_config\_t::intEnable**

17.2.1.1.0.40.7 **pdb\_mult\_factor\_mode\_t pdb\_user\_config\_t::multFactorMode**

17.2.1.1.0.40.8 **bool pdb\_user\_config\_t::continuousModeEnable**

17.2.1.1.0.40.9 **uint32\_t pdb\_user\_config\_t::pdbModulusValue**

17.2.1.1.0.40.10 **uint32\_t pdb\_user\_config\_t::delayValue**

#### 17.2.1.2 struct pdb\_adc\_pre\_trigger\_config\_t

This structure keeps the configuration for ADC pre-trigger in PDB module.

#### Data Fields

- **bool backToBackModeEnable**  
*Switch to enable the back-to-back mode.*
- **bool preTriggerOutEnable**  
*Switch to enable trigger out the pre-channel.*
- **uint32\_t delayValue**  
*Set the value for ADC pre-trigger's delay value.*

## PDB Peripheral Driver

### 17.2.1.2.0.41 Field Documentation

17.2.1.2.0.41.1 `bool pdb_adc_pre_trigger_config_t::backToBackModeEnable`

17.2.1.2.0.41.2 `bool pdb_adc_pre_trigger_config_t::preTriggerOutEnable`

17.2.1.2.0.41.3 `uint32_t pdb_adc_pre_trigger_config_t::delayValue`

### 17.2.1.3 `struct pdb_dac_trigger_config_t`

This structure keeps the configuration for DAC trigger in the PDB module.

#### Data Fields

- `bool extTriggerInputEnable`  
*Switch to enable the external trigger for DAC interval counter.*

### 17.2.1.3.0.42 Field Documentation

17.2.1.3.0.42.1 `bool pdb_dac_trigger_config_t::extTriggerInputEnable`

### 17.2.1.4 `struct pdb_pulse_out_trigger_config_t`

This structure keeps the configuration for the pulse-out trigger in the PDB module.

#### Data Fields

- `uint32_t pulseOutHighValue`  
*Set the value for that pulse out goes high when the PDB counter reaches to this value.*
- `uint32_t pulseOutLowValue`  
*Set the value for that pulse out goes low when the PDB counter reaches to this value.*

### 17.2.1.4.0.43 Field Documentation

17.2.1.4.0.43.1 `uint32_t pdb_pulse_out_trigger_config_t::pulseOutHighValue`

17.2.1.4.0.43.2 `uint32_t pdb_pulse_out_trigger_config_t::pulseOutLowValue`

### 17.2.1.5 `struct pdb_state_t`

The contents of this structure are internal to the driver and should not be modified by users.

#### Data Fields

- `pdb_callback_t userCallbackFunc`  
*Keep the user-defined callback function.*

### 17.2.1.5.0.44 Field Documentation

#### 17.2.1.5.0.44.1 `pdb_callback_t pdb_state_t::userCallbackFunc`

### 17.2.2 Enumeration Type Documentation

#### 17.2.2.1 `enum pdb_adc_pre_trigger_flag_t`

Enumerator

`kPdbAdcPreChnFlag` ADC pre-trigger flag when the counter reaches to pre-trigger's delay value.

`kPdbAdcPreChnErrFlag` ADC pre-trigger sequence error flag.

### 17.2.3 Function Documentation

#### 17.2.3.1 `pdb_status_t PDB_DRV_StructInitUserConfigForSoftTrigger ( pdb_user_config_t * userConfigPtr )`

This function fills the initial user configuration. It is set as one-time, software trigger mode. The PDB modulus and delay value are all set to maximum. The PDB interrupt is enabled and causes the PDB interrupt when the counter hits the delay value. Then, the PDB module is initialized as a normal timer if no other setting is changed. The settings are:

```
.loadMode = kPdbLoadImmediately; .seqErrIntEnable = false; .dmaEnable = false; .clkPrescalerDivMode = kPdbClkPreDivBy128; .triggerSrcMode = kPdbSoftTrigger; .intEnable = true; .multFactorMode = kPdbMultFactorAs40; .continuousModeEnable = false; .pdbModulusValue = 0xFFFFU; .delayValue = 0xFFFFU;
```

Parameters

|                            |                                                                           |
|----------------------------|---------------------------------------------------------------------------|
| <code>userConfigPtr</code> | Pointer to the user configuration structure. See the "pdb_user_config_t". |
|----------------------------|---------------------------------------------------------------------------|

Returns

Execution status.

#### 17.2.3.2 `pdb_status_t PDB_DRV_Init ( uint32_t instance, pdb_user_config_t * userConfigPtr, pdb_state_t * userStatePtr )`

This function initializes the PDB counter and triggers the input. It resets PDB registers and enables the PDB clock. Therefore, it should be called before any other operation. After it is initialized, the PDB can act as a triggered timer, which enables other features in PDB module.

## PDB Peripheral Driver

Parameters

|                      |                                                                                |
|----------------------|--------------------------------------------------------------------------------|
| <i>instance</i>      | PDB instance ID.                                                               |
| <i>userConfigPtr</i> | Pointer to the user configuration structure. See the "pdb_user_config_t".      |
| <i>userStatePtr</i>  | Pointer to the structure for keeping an internal state. See the "pdb_state_t". |

Returns

Execution status.

### 17.2.3.3 void PDB\_DRV\_Deinit ( uint32\_t *instance* )

This function de-initializes the PDB module. Calling this function shuts down the PDB module and reduces the power consumption.

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | PDB instance ID. |
|-----------------|------------------|

### 17.2.3.4 void PDB\_DRV\_SoftTriggerCmd ( uint32\_t *instance* )

This function triggers the PDB with a software trigger. When the PDB is set to use the software trigger as input, calling this function triggers the PDB.

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | PDB instance ID. |
|-----------------|------------------|

### 17.2.3.5 uint32\_t PDB\_DRV\_GetCurrentCounter ( uint32\_t *instance* )

This function gets the current counter value.

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | PDB instance ID. |
|-----------------|------------------|

Returns

Current PDB counter value.

### 17.2.3.6 **bool PDB\_DRV\_GetPdbCounterIntFlag ( uint32\_t *instance* )**

This function gets the PDB interrupt flag. It is asserted if the PDB interrupt occurs.

## PDB Peripheral Driver

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | PDB instance ID. |
|-----------------|------------------|

Returns

Assertion of indicated event.

### 17.2.3.7 void PDB\_DRV\_ClearPdbCounterIntFlag ( uint32\_t *instance* )

This function clears the interrupt flag.

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | PDB instance ID. |
|-----------------|------------------|

### 17.2.3.8 pdb\_status\_t PDB\_DRV\_EnableAdcPreTrigger ( uint32\_t *instance*,                   uint32\_t *adcChn*, uint32\_t *preChn*, pdb\_adc\_pre\_trigger\_config\_t \*                   *adcPreTriggerConfigPtr* )

This function enables the ADC pre-trigger with its configuration. The setting value takes effect according to the load-mode configured during the PDB initialization, although the load operation was done inside the function. This is an additional function based on the PDB counter.

Parameters

|                               |                                                                               |
|-------------------------------|-------------------------------------------------------------------------------|
| <i>instance</i>               | PDB instance ID.                                                              |
| <i>adcChn</i>                 | ADC trigger channel, related to the ADC instance.                             |
| <i>preChn</i>                 | ADC pre-trigger channel, related to the ADC channel group instance.           |
| <i>adcPreTriggerConfigPtr</i> | Pointer to structure of configuration, see to "pdb_adc_pre_trigger_config_t". |

Returns

Execution status.

### 17.2.3.9 void PDB\_DRV\_DisableAdcPreTrigger ( uint32\_t *instance*, uint32\_t *adcChn*,                   uint32\_t *preChn* )

This function disables the ADC pre-trigger. The PDB works the same way as when the pre-trigger was not enabled.

Parameters

|                 |                                                                     |
|-----------------|---------------------------------------------------------------------|
| <i>instance</i> | PDB instance ID.                                                    |
| <i>adcChn</i>   | ADC trigger channel, related to the ADC instance.                   |
| <i>preChn</i>   | ADC pre-trigger channel, related to the ADC channel group instance. |

#### 17.2.3.10 **bool PDB\_DRV\_GetAdcPreTriggerFlag ( uint32\_t *instance*, uint32\_t *adcChn*, uint32\_t *preChn*, pdb\_adc\_pre\_trigger\_flag\_t *flag* )**

This function gets the pre-trigger flag for the PDB module. It is asserted if a related event occurs.

Parameters

|                 |                                                                     |
|-----------------|---------------------------------------------------------------------|
| <i>instance</i> | PDB instance ID.                                                    |
| <i>adcChn</i>   | ADC trigger channel, related to the ADC instance.                   |
| <i>preChn</i>   | ADC pre-trigger channel, related to the ADC channel group instance. |
| <i>flag</i>     | Indicated flag for event.                                           |

Returns

Assertion of indicated event.

#### 17.2.3.11 **void PDB\_DRV\_ClearAdcPreTriggerFlag ( uint32\_t *instance*, uint32\_t *adcChn*, uint32\_t *preChn*, pdb\_adc\_pre\_trigger\_flag\_t *flag* )**

This function clears the ADC pre-trigger flag for the PDB module.

Parameters

|                 |                                                                     |
|-----------------|---------------------------------------------------------------------|
| <i>instance</i> | PDB instance ID.                                                    |
| <i>adcChn</i>   | ADC trigger channel, related to the ADC instance.                   |
| <i>preChn</i>   | ADC pre-trigger channel, related to the ADC channel group instance. |
| <i>flag</i>     | Indicated flag for event.                                           |

#### 17.2.3.12 **pdb\_status\_t PDB\_DRV\_EnableDacTrigger ( uint32\_t *instance*, uint32\_t *dacChn*, pdb\_dac\_trigger\_config\_t \* *dacTriggerConfigPtr* )**

This function enables the DAC trigger with its configuration. The setting value takes effect according to the load mode configured during PDB initialization, although the load operation is done inside the function. This is an additional function based on the PDB counter.

## PDB Peripheral Driver

Parameters

|                            |                                                                           |
|----------------------------|---------------------------------------------------------------------------|
| <i>instance</i>            | PDB instance ID.                                                          |
| <i>dacChn</i>              | DAC trigger channel, related to the DAC instance.                         |
| <i>dacTriggerConfigPtr</i> | Pointer to structure of configuration, see to "pdb_dac_trigger_config_t". |

Returns

Execution status.

### 17.2.3.13 void PDB\_DRV\_DisableDacTrigger ( uint32\_t *instance*, uint32\_t *dacChn* )

This function disables the DAC trigger. The PDB works the same as when the DAC trigger was not enabled.

Parameters

|                 |                                                   |
|-----------------|---------------------------------------------------|
| <i>instance</i> | PDB instance ID.                                  |
| <i>dacChn</i>   | DAC trigger channel, related to the DAC instance. |

### 17.2.3.14 pdb\_status\_t PDB\_DRV\_EnablePulseOutTrigger ( uint32\_t *instance*, uint32\_t *pulseChn*, pdb\_pulse\_out\_trigger\_config\_t \* *pulseOutTriggerConfigPtr* )

This function enables the pulse-out trigger with its configuration. The setting value takes effect according to the load mode configured during the PDB initialization, although the load operation is done inside the function. This is an additional function based on the PDB counter.

Parameters

|                                 |                                                                                 |
|---------------------------------|---------------------------------------------------------------------------------|
| <i>instance</i>                 | PDB instance ID.                                                                |
| <i>pulseChn</i>                 | Pulse out trigger channel.                                                      |
| <i>pulseOutTriggerConfigPtr</i> | Pointer to structure of configuration, see to "pdb_pulse_out_trigger_config_t". |

Returns

Execution status.

**17.2.3.15 void PDB\_DRV\_DisablePulseOutTrigger ( uint32\_t *instance*, uint32\_t *pulseChn* )**

This function disables the pulse-out trigger. The PDB works the same as when the pulse out-trigger was not been enabled.

## PDB Peripheral Driver

Parameters

|                 |                            |
|-----------------|----------------------------|
| <i>instance</i> | PDB instance ID.           |
| <i>pulseChn</i> | Pulse out trigger channel. |

### 17.2.3.16 **pdb\_status\_t PDB\_DRV\_InstallCallback ( uint32\_t *instance*, pdb\_callback\_t *userCallback* )**

This function installs the user-defined callback. When the PDB interrupt request is served, the callback is executed inside the ISR.

Parameters

|                     |                                 |
|---------------------|---------------------------------|
| <i>instance</i>     | PDB instance ID.                |
| <i>userCallback</i> | User-defined callback function. |

Returns

Execution status.

### 17.2.3.17 **void PDB\_DRV\_IRQHandler ( uint32\_t *instance* )**

This function is the driver-defined ISR in the PDB module. It includes the process for interrupt mode defined by the driver. Currently, it is called inside the system-defined ISR.

Parameters

|                 |                  |
|-----------------|------------------|
| <i>instance</i> | PDB instance ID. |
|-----------------|------------------|

# Chapter 18

## Periodic Interrupt Timer (PIT)

The Kinetis SDK provides both HAL and Peripheral drivers for the Periodic Interrupt Timer (PIT) block of Kinetis devices.

### Modules

- [PIT HAL driver](#)

*This part describes the programming interface of the PIT HAL driver.*

- [PIT ISR Definitions](#)

*This part describes the PIT interrupt definitions.*

- [PIT Peripheral Driver](#)

*This part describes the programming interface of the PIT Peripheral driver.*

## PIT HAL driver

### 18.1 PIT HAL driver

This chapter describes the programming interface of the PIT HAL driver.

#### Initialization

- static void [PIT\\_HAL\\_Enable](#) (uint32\_t baseAddr)  
*Enables the PIT module.*
- static void [PIT\\_HAL\\_Disable](#) (uint32\_t baseAddr)  
*Disables the PIT module.*
- static void [PIT\\_HAL\\_SetTimerRunInDebugCmd](#) (uint32\_t baseAddr, bool timerRun)  
*Configures the timers to continue running or to stop in debug mode.*

#### Timer Start and Stop

- static void [PIT\\_HAL\\_StartTimer](#) (uint32\_t baseAddr, uint32\_t channel)  
*Starts the timer counting.*
- static void [PIT\\_HAL\\_StopTimer](#) (uint32\_t baseAddr, uint32\_t channel)  
*Stops the timer from counting.*
- static bool [PIT\\_HAL\\_IsTimerRunning](#) (uint32\_t baseAddr, uint32\_t channel)  
*Checks to see whether the current timer is started or not.*

#### Timer Period

- static void [PIT\\_HAL\\_SetTimerPeriodByCount](#) (uint32\_t baseAddr, uint32\_t channel, uint32\_t count)  
*Sets the timer period in units of count.*
- static uint32\_t [PIT\\_HAL\\_GetTimerPeriodByCount](#) (uint32\_t baseAddr, uint32\_t channel)  
*Returns the current timer period in units of count.*
- static uint32\_t [PIT\\_HAL\\_ReadTimerCount](#) (uint32\_t baseAddr, uint32\_t channel)  
*Reads the current timer counting value.*

#### Interrupt

- static void [PIT\\_HAL\\_SetIntCmd](#) (uint32\_t baseAddr, uint32\_t channel, bool enable)  
*Enables or disables the timer interrupt.*
- static bool [PIT\\_HAL\\_GetIntCmd](#) (uint32\_t baseAddr, uint32\_t channel)  
*Checks whether the timer interrupt is enabled or not.*
- static void [PIT\\_HAL\\_ClearIntFlag](#) (uint32\_t baseAddr, uint32\_t channel)  
*Clears the timer interrupt flag.*
- static bool [PIT\\_HAL\\_IsIntPending](#) (uint32\_t baseAddr, uint32\_t channel)  
*Reads the current timer timeout flag.*

### 18.1.0.18 PIT HAL Driver

#### Overview

PIT HAL driver is a set of API functions used to access and configure the PIT hardware registers.

#### 18.1.1 Function Documentation

##### 18.1.1.1 static void PIT\_HAL\_Enable ( *uint32\_t baseAddr* ) [inline], [static]

This function enables the PIT timer clock (Note: this function does not un-gate the system clock gating control). It should be called before any other timer related setup.

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
|-----------------|----------------------------------------|

##### 18.1.1.2 static void PIT\_HAL\_Disable ( *uint32\_t baseAddr* ) [inline], [static]

This function disables all PIT timer clocks (Note: it does not affect the SIM clock gating control).

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
|-----------------|----------------------------------------|

##### 18.1.1.3 static void PIT\_HAL\_SetTimerRunInDebugCmd ( *uint32\_t baseAddr*, *bool timerRun* ) [inline], [static]

In debug mode, the timers may or may not be frozen, based on the configuration of this function. This is intended to aid software development, allowing the developer to halt the processor, investigate the current state of the system (for example, the timer values), and continue the operation.

Parameters

|                 |                                                                                                                                                                                |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance.                                                                                                                                         |
| <i>timerRun</i> | Timers run or stop in debug mode. <ul style="list-style-type: none"> <li>• true: Timers continue to run in debug mode.</li> <li>• false: Timers stop in debug mode.</li> </ul> |

## PIT HAL driver

### 18.1.1.4 static void PIT\_HAL\_StartTimer ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]

After calling this function, timers load the start value as specified by the function [PIT\\_HAL\\_SetTimerPeriodByCount\(uint32\\_t baseAddr, uint32\\_t channel, uint32\\_t count\)](#), count down to 0, and load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the time-out interrupt flag.

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

### 18.1.1.5 static void PIT\_HAL\_StopTimer ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]

This function stops every timer from counting. Timers reload their periods respectively after they call the PIT\_HAL\_StartTimer the next time.

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

### 18.1.1.6 static bool PIT\_HAL\_IsTimerRunning ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

Returns

Current timer running status -true: Current timer is running. -false: Current timer has stopped.

### 18.1.1.7 static void PIT\_HAL\_SetTimerPeriodByCount ( uint32\_t *baseAddr*, uint32\_t *channel*, uint32\_t *count* ) [inline], [static]

Timers begin counting from the value set by this function. The counter period of a running timer can be modified by first stopping the timer, setting a new load value, and starting the timer again. If timers are not restarted, the new value is loaded after the next trigger event.

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |
| <i>count</i>    | Timer period in units of count         |

#### **18.1.1.8 static uint32\_t PIT\_HAL\_GetTimerPeriodByCount ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]**

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

Returns

Timer period in units of count

#### **18.1.1.9 static uint32\_t PIT\_HAL\_ReadTimerCount ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]**

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

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

Returns

Current timer counting value

#### **18.1.1.10 static void PIT\_HAL\_SetIntCmd ( uint32\_t *baseAddr*, uint32\_t *channel*, bool *enable* ) [inline], [static]**

If enabled, an interrupt happens when a timeout event occurs (Note: NVIC should be called to enable pit interrupt in system level).

## PIT HAL driver

Parameters

|                 |                                                                                                                                                                             |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance.                                                                                                                                      |
| <i>channel</i>  | Timer channel number                                                                                                                                                        |
| <i>enable</i>   | Enable or disable interrupt. <ul style="list-style-type: none"><li>• true: Generate interrupt when timer counts to 0.</li><li>• false: No interrupt is generated.</li></ul> |

### 18.1.1.11 static bool PIT\_HAL\_GetIntCmd ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

Returns

Status of enabled or disabled interrupt

- true: Interrupt is enabled.
- false: Interrupt is disabled.

### 18.1.1.12 static void PIT\_HAL\_ClearIntFlag ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]

This function clears the timer interrupt flag after a timeout event occurs.

Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

### 18.1.1.13 static bool PIT\_HAL\_IsIntPending ( uint32\_t *baseAddr*, uint32\_t *channel* ) [inline], [static]

Every time the timer counts to 0, this flag is set.

## Parameters

|                 |                                        |
|-----------------|----------------------------------------|
| <i>baseAddr</i> | Base address for current PIT instance. |
| <i>channel</i>  | Timer channel number                   |

## Returns

Current status of the timeout flag

- true: Timeout has occurred.
- false: Timeout has not yet occurred.

### 18.2 PIT Peripheral Driver

This chapter describes the programming interface of the PIT Peripheral driver.

### Data Structures

- struct [pit\\_user\\_config\\_t](#)  
*PIT timer configuration structure. [More...](#)*

### TypeDefs

- typedef void(\* [pit\\_isr\\_callback\\_t](#) )(void)  
*PIT ISR callback function typedef.*

### Initialize and Shutdown

- void [PIT\\_DRV\\_Init](#) (uint32\_t instance, bool isRunInDebug)  
*Initializes the PIT module.*
- void [PIT\\_DRV\\_Deinit](#) (uint32\_t instance)  
*Disables the PIT module and gate control.*
- void [PIT\\_DRV\\_InitChannel](#) (uint32\_t instance, uint32\_t channel, const [pit\\_user\\_config\\_t](#) \*config)  
*Initializes the PIT channel.*

### Timer Start and Stop

- void [PIT\\_DRV\\_StartTimer](#) (uint32\_t instance, uint32\_t channel)  
*Starts the timer counting.*
- void [PIT\\_DRV\\_StopTimer](#) (uint32\_t instance, uint32\_t channel)  
*Stops the timer counting.*

### Timer Period

- void [PIT\\_DRV\\_SetTimerPeriodByUs](#) (uint32\_t instance, uint32\_t channel, uint32\_t us)  
*Sets the timer period in microseconds.*
- uint32\_t [PIT\\_DRV\\_ReadTimerUs](#) (uint32\_t instance, uint32\_t channel)  
*Reads the current timer value in microseconds.*

### ISR Callback Function

- void [PIT\\_DRV\\_InstallCallback](#) (uint32\_t instance, uint32\_t channel, [pit\\_isr\\_callback\\_t](#) function)  
*Registers the PIT ISR callback function.*

### 18.2.0.14 PIT Peripheral Driver

#### Overview

The PIT driver configures PIT timers and provides an easy way to initialize and configure timer periods.

#### Initialization

To initialize the PIT module, call the `PIT_DRV_Init` function. This function enables the PIT module and clock automatically. The parameter passed in the `PIT_DRV_Init` configures the timers to run or stop in debug mode. To use one timer channel, call the `PIT_DRV_InitChannel` initialize that channel.

This is example code to initialize and configure the driver:

```
// Define device configuration.
const pit_config_t pitInit = {
 isInterruptEnabled = false, // Disable timer interrupt.
 isTimerChained = false, // Meaningless for timer 0.
 periodUs = 20U // Set timer period to 20 us.
};

// Initialize PIT instance 0. Timers will stop running in debug mode.
PIT_DRV_Init(0, stop);

// Initialize PIT instance 0, timer 0.
PIT_DRV_InitChannel(0, 0, &pitInit);
```

#### Timer Period

The PIT driver provides four ways to set the timer period.

1. The `PIT_DRV_InitChannel` function sets the timer period in microseconds. It is only applicable when initializing the channel.
2. The void `PIT_DRV_SetTimerPeriodByUs(uint32_t instance, uint32_t timer, uint32_t us)` function sets the timer period in microseconds. It is applicable at any time.
3. The void `PIT_DRV_SetLifetimeTimerPeriodByUs(uint32_t instance, uint64_t us)` function sets the lifetime timer period in microseconds. It only supports specific MCUs. Check the appropriate reference manual before using this function.
4. The void `PIT_HAL_SetTimerPeriodByCount(uint32_t instance, uint32_t timer, uint32_t count)` function sets the timer period in units of count. To use this function, include the `fsl_pit_hal.h`.

To read the current timer counting value in microseconds, call the `uint32_t PIT_DRV_ReadTimerUs(uint32_t instance, uint32_t timer)` function.

#### Timer Operation

After the timer setting is complete, call the void `PIT_DRV_StartTimer(uint32_t timer)` function to start timer counting. Call the void `PIT_DRV_StopTimer(uint32_t instance, uint32_t timer)` function to stop it

## PIT Peripheral Driver

at any time.

If you want to close the PIT module entirely, call the void [PIT\\_DRV\\_Deinit\(uint32\\_t instance\)](#) function. This disables both the PIT module and the clock gate.

### 18.2.1 Data Structure Documentation

#### 18.2.1.1 `struct pit_user_config_t`

Define structure PitConfig and use the [PIT\\_DRV\\_InitChannel\(\)](#) function to make necessary initializations. You may also use the remaining functions for PIT configuration.

Note

The timer chain feature is not valid in all devices. Check the `fsl_pit_features.h` for accurate settings. If it's not valid, the value set here will be bypassed inside the [PIT\\_DRV\\_InitChannel\(\)](#) function.

#### Data Fields

- bool `isInterruptEnabled`  
*Timer interrupt 0-disable/1-enable.*
- bool `isTimerChained`  
*Chained with previous timer, 0-not/1-chained.*
- `uint32_t periodUs`  
*Timer period in unit of microseconds.*

### 18.2.2 Function Documentation

#### 18.2.2.1 `void PIT_DRV_Init( uint32_t instance, bool isRunInDebug )`

This function must be called before calling all the other PIT driver functions. This function un-gates the PIT clock and enables the PIT module. The `isRunInDebug` passed into function affects all timer channels.

Parameters

|                           |                                                                                                                                                                             |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>instance</code>     | PIT module instance number.                                                                                                                                                 |
| <code>isRunInDebug</code> | Timers run or stop in debug mode. <ul style="list-style-type: none"><li>• true: Timers continue to run in debug mode.</li><li>• false: Timers stop in debug mode.</li></ul> |

### 18.2.2.2 void PIT\_DRV\_Deinit ( uint32\_t *instance* )

This function disables all PIT interrupts and PIT clock. It then gates the PIT clock control. PIT\_DRV\_Init must be called if you want to use PIT again.

## PIT Peripheral Driver

Parameters

|                 |                             |
|-----------------|-----------------------------|
| <i>instance</i> | PIT module instance number. |
|-----------------|-----------------------------|

### 18.2.2.3 void PIT\_DRV\_InitChannel ( *uint32\_t instance, uint32\_t channel, const pit\_user\_config\_t \* config* )

This function initializes the PIT timers by using a channel. Pass in the timer number and its configuration structure. Timers do not start counting by default after calling this function. The function PIT\_DRV\_StartTimer must be called to start the timer counting. Call the PIT\_DRV\_SetTimerPeriodByUs to re-set the period.

This is an example demonstrating how to define a PIT channel configuration structure:

```
pit_user_config_t pitTestInit = {
 .isInterruptEnabled = true,
 // Only takes effect when chain feature is available.
 // Otherwise, pass in arbitrary value(true/false).
 .isTimerChained = false,
 // In unit of microseconds.
 .periodUs = 1000,
};
```

Parameters

|                 |                                      |
|-----------------|--------------------------------------|
| <i>instance</i> | PIT module instance number.          |
| <i>channel</i>  | Timer channel number.                |
| <i>config</i>   | PIT channel configuration structure. |

### 18.2.2.4 void PIT\_DRV\_StartTimer ( *uint32\_t instance, uint32\_t channel* )

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

|                 |                             |
|-----------------|-----------------------------|
| <i>instance</i> | PIT module instance number. |
| <i>channel</i>  | Timer channel number.       |

### 18.2.2.5 void PIT\_DRV\_StopTimer ( *uint32\_t instance, uint32\_t channel* )

This function stops every timer counting. Timers reload their periods respectively after the next time they call the PIT\_DRV\_StartTimer.

Parameters

|                 |                             |
|-----------------|-----------------------------|
| <i>instance</i> | PIT module instance number. |
| <i>channel</i>  | Timer channel number.       |

#### 18.2.2.6 void PIT\_DRV\_SetTimerPeriodByUs ( *uint32\_t instance, uint32\_t channel, uint32\_t us* )

The period range depends on the frequency of the PIT source clock. If the required period is out of range, use the lifetime timer.

Parameters

|                 |                               |
|-----------------|-------------------------------|
| <i>instance</i> | PIT module instance number.   |
| <i>channel</i>  | Timer channel number.         |
| <i>us</i>       | Timer period in microseconds. |

#### 18.2.2.7 uint32\_t PIT\_DRV\_ReadTimerUs ( *uint32\_t instance, uint32\_t channel* )

This function returns an absolute time stamp in microseconds. One common use of this function is to measure the running time of a part of code. Call this function at both the beginning and end of code. The time difference between these two time stamps is the running time. Make sure the running time does not exceed the timer period. The time stamp returned is up-counting.

Parameters

|                 |                             |
|-----------------|-----------------------------|
| <i>instance</i> | PIT module instance number. |
| <i>channel</i>  | Timer channel number.       |

Returns

Current timer value in microseconds.

#### 18.2.2.8 void PIT\_DRV\_InstallCallback ( *uint32\_t instance, uint32\_t channel, pit\_isr\_callback\_t function* )

System default ISR interfaces are already defined in the `fsl_pit_irq.c`. Users can either edit these ISRs or use this function to register a callback function. The default ISR runs the callback function if there is one installed.

## PIT Peripheral Driver

### Parameters

|                 |                                       |
|-----------------|---------------------------------------|
| <i>instance</i> | PIT module instance number.           |
| <i>channel</i>  | Timer channel number.                 |
| <i>function</i> | Pointer to pit ISR callback function. |

### **18.3 PIT ISR Definitions**

This chapter describes the PIT interrupt definitions.

## PIT ISR Definitions

# Chapter 19

## Random Number Generator Accelerator (RNGA)

The Kinetis SDK provides both HAL and Peripheral drivers for the Random Number Generator Accelerator (RNGA) block of Kinetis devices.

### Modules

- [RNGA HAL driver](#)

*This part describes the programming interface of the RNGA HAL driver.*

- [RNGA Peripheral Driver](#)

*This part describes the programming interface of the RNGA Peripheral driver.*

### 19.1 RNGA HAL driver

This chapter describes the programming interface of the RNGA HAL driver.

#### Enumerations

- enum `rnga_mode_t`  
*RNGA working mode.*
- enum `rnga_output_reg_level_t`  
*Defines the value of output register level.*

#### RNGA HAL.

- static void `RNGA_HAL_Init` (uint32\_t rngaBaseAddr)  
*Initializes the RNGA module.*
- static void `RNGA_HAL_Enable` (uint32\_t rngaBaseAddr)  
*Enables the RNGA module.*
- static void `RNGA_HAL_Disable` (uint32\_t rngaBaseAddr)  
*Disables the RNGA module.*
- static void `RNGA_HAL_SetHighAssuranceCmd` (uint32\_t rngaBaseAddr, bool enable)  
*Sets the RNGA high assurance.*
- static void `RNGA_HAL_SetIntMaskCmd` (uint32\_t rngaBaseAddr, bool enable)  
*Sets the RNGA interrupt mask.*
- static void `RNGA_HAL_ClearIntCmd` (uint32\_t rngaBaseAddr, bool enable)  
*Clears the RNGA interrupt.*
- static void `RNGA_HAL_SetWorkModeCmd` (uint32\_t rngaBaseAddr, `rnga_mode_t` mode)  
*Sets the RNGA in sleep mode or normal mode.*
- static `rnga_output_reg_level_t RNGA_HAL_GetOutputRegSize` (uint32\_t rngaBaseAddr)  
*Gets the output register size.*
- static `rnga_output_reg_level_t RNGA_HAL_GetOutputRegLevel` (uint32\_t rngaBaseAddr)  
*Gets the output register level.*
- static `rnga_mode_t RNGA_HAL_GetWorkMode` (uint32\_t rngaBaseAddr)  
*Gets the RNGA working mode.*
- static bool `RNGA_HAL_GetErrorIntCmd` (uint32\_t rngaBaseAddr)  
*Gets the RNGA status whether an error interrupt has occurred.*
- static bool `RNGA_HAL_GetOutputRegUnderflowCmd` (uint32\_t rngaBaseAddr)  
*Gets the RNGA status whether an output register underflow has occurred.*
- static bool `RNGA_HAL_GetLastReadStatusCmd` (uint32\_t rngaBaseAddr)  
*Gets the most recent RNGA read status.*
- static bool `RNGA_HAL_GetSecurityViolationCmd` (uint32\_t rngaBaseAddr)  
*Gets the RNGA status whether a security violation has occurred.*
- static uint32\_t `RNGA_HAL_ReadRandomData` (uint32\_t rngaBaseAddr)  
*Gets a random data from the RNGA.*
- static void `RNGA_HAL_WriteSeed` (uint32\_t rngaBaseAddr, uint32\_t data)  
*Inputs an entropy value used to seed the RNGA.*

### 19.1.1 Function Documentation

#### 19.1.1.1 static void RNGA\_HAL\_Init( uint32\_t *rngaBaseAddr* ) [inline], [static]

This function initializes the RNGA to a default state.

## RNGA HAL driver

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

### 19.1.1.2 static void RNGA\_HAL\_Enable ( uint32\_t *rngaBaseAddr* ) [inline], [static]

This function enables the RNGA random data generation and loading.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

### 19.1.1.3 static void RNGA\_HAL\_Disable ( uint32\_t *rngaBaseAddr* ) [inline], [static]

This function disables the RNGA module.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

### 19.1.1.4 static void RNGA\_HAL\_SetHighAssuranceCmd ( uint32\_t *rngaBaseAddr*, bool *enable* ) [inline], [static]

This function sets the RNGA high assurance(notification of security violations).

Parameters

|                                |                                                                                                          |
|--------------------------------|----------------------------------------------------------------------------------------------------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address                                                                                             |
| <i>enable,0</i>                | means notification of security violations disabled. 1 means notification of security violations enabled. |

### 19.1.1.5 static void RNGA\_HAL\_SetIntMaskCmd ( uint32\_t *rngaBaseAddr*, bool *enable* ) [inline], [static]

This function sets the RNGA error interrupt mask.

Parameters

|                                |                                                           |
|--------------------------------|-----------------------------------------------------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address                                              |
| <i>enable,0</i>                | means unmask RNGA interrupt. 1 means mask RNGA interrupt. |

#### 19.1.1.6 static void RNGA\_HAL\_ClearIntCmd ( uint32\_t *rngaBaseAddr*, bool *enable* ) [inline], [static]

This function clears the RNGA interrupt.

Parameters

|                                |                                                                |
|--------------------------------|----------------------------------------------------------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address                                                   |
| <i>enable,0</i>                | means do not clear the interrupt. 1 means clear the interrupt. |

#### 19.1.1.7 static void RNGA\_HAL\_SetWorkModeCmd ( uint32\_t *rngaBaseAddr*, *rnga\_mode\_t mode* ) [inline], [static]

This function specifies whether the RNGA is in sleep mode or normal mode.

Parameters

|                                     |                                                                               |
|-------------------------------------|-------------------------------------------------------------------------------|
| <i>rngaBase-<br/>Addr,RNGA</i>      | base address                                                                  |
| <i>mode,kMode_-<br/>RNGA_Normal</i> | means set RNGA in normal mode. kMode_RNGA_Sleep means set RNGA in sleep mode. |

#### 19.1.1.8 static *rnga\_output\_reg\_level\_t* RNGA\_HAL\_GetOutputRegSize ( uint32\_t *rngaBaseAddr* ) [inline], [static]

This function gets the size of the output register as 32-bit random data words it can hold.

Parameters

## RNGA HAL driver

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

1 means one word(this value is fixed).

### 19.1.1.9 static **rnga\_output\_reg\_level\_t RNGA\_HAL\_GetOutputRegLevel ( uint32\_t rngaBaseAddr ) [inline], [static]**

This function gets the number of random-data words that are in OR [RANDOUT], which indicates if OR is valid.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

0 means no words(empty), 1 means one word(valid).

### 19.1.1.10 static **rnga\_mode\_t RNGA\_HAL\_GetWorkMode ( uint32\_t rngaBaseAddr ) [inline], [static]**

This function checks whether the RNGA works in sleep mode or normal mode.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

Kmode\_RNGA\_Normal means in normal mode Kmode\_RNGA\_Sleep means in sleep mode

### 19.1.1.11 static **bool RNGA\_HAL\_GetErrorIntCmd ( uint32\_t rngaBaseAddr ) [inline], [static]**

This function gets the RNGA status whether an OR underflow condition has occurred since the error interrupt was last cleared or the RNGA was reset.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

0 means no underflow, 1 means underflow

#### **19.1.1.12 static bool RNGA\_HAL\_GetOutputRegUnderflowCmd ( uint32\_t *rngaBaseAddr* ) [inline], [static]**

This function gets the RNGA status whether an OR underflow condition has occurred since the register (SR) was last read or the RNGA was reset.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

0 means no underflow, 1 means underflow

#### **19.1.1.13 static bool RNGA\_HAL\_GetLastReadStatusCmd ( uint32\_t *rngaBaseAddr* ) [inline], [static]**

This function gets the RNGA status whether the most recent read of OR[RANDOUT] causes an OR underflow condition.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

0 means no underflow, 1 means underflow

#### **19.1.1.14 static bool RNGA\_HAL\_GetSecurityViolationCmd ( uint32\_t *rngaBaseAddr* ) [inline], [static]**

This function gets the RNGA status whether a security violation has occurred when high assurance is enabled.

## RNGA HAL driver

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

0 means no security violation, 1 means security violation

### 19.1.1.15 static uint32\_t RNGA\_HAL\_ReadRandomData ( uint32\_t *rngaBaseAddr* ) [inline], [static]

This function gets a random data from RNGA.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

Returns

random data obtained

### 19.1.1.16 static void RNGA\_HAL\_WriteSeed ( uint32\_t *rngaBaseAddr*, uint32\_t *data* ) [inline], [static]

This function specifies an entropy value that RNGA uses with its ring oscillations to seed its pseudorandom algorithm.

Parameters

|                                |              |
|--------------------------------|--------------|
| <i>rngaBase-<br/>Addr,RNGA</i> | base address |
|--------------------------------|--------------|

## 19.2 RNGA Peripheral Driver

This chapter describes the programming interface of the RNGA Peripheral driver.

### Data Structures

- struct [rnga\\_user\\_config\\_t](#)  
*Data structure for the RNGA initialization. [More...](#)*

### Enumerations

- enum [rnga\\_status\\_t](#)  
*Status structure for RNGA.*

### RNGA Driver

- [rnga\\_status\\_t RNGA\\_DRV\\_Init](#) (uint32\_t instance, [rnga\\_user\\_config\\_t](#) \*config)
- void [RNGA\\_DRV\\_Deinit](#) (uint32\_t instance)
- void [RNGA\\_DRV\\_SetMode](#) (uint32\_t instance, [rnga\\_mode\\_t](#) mode)
- [rnga\\_mode\\_t RNGA\\_DRV\\_GetMode](#) (uint32\_t instance)
- [rnga\\_status\\_t RNGA\\_DRV\\_GetRandomData](#) (uint32\_t instance, uint32\_t \*data)
- void [RNGA\\_DRV\\_Seed](#) (uint32\_t instance, uint32\_t seed)
- void [RNGA\\_DRV\\_IRQHandler](#) (uint32\_t instance)

#### 19.2.0.17 RNGA Peripheral Driver

### Overview

The RNGA driver provides an easy way to initialize and configure the RNGA.

### Initialization

To initialize the RNGA module, call the `RNGA_DRV_Init()` function and pass in the user configuration structure. This function automatically enables the RNGA module and clock.

After the `RNGA_DRV_Init()` function is called, the RNGA is enabled and its counter is working.

This example code shows how to initialize and configure the driver:

```
// Define device configuration.
const rnga_user_config_t init =
{
 .isIntMasked = true, /* Mask RNGA Error Interrupt */
 .highAssuranceEnable = true, /* Enable high assurance */
};

// Initialize RNGA.
RNGA_DRV_Init(&init);
```

### RNGA Set/Get Working Mode

The RNGA works either in sleep mode or normal mode

1. RNGA\_DRV\_SetMode() Set RNGA mode.
2. RNGA\_DRV\_GetMode() Get RNGA working mode.

### Get random data from RNGA

1. RNGA\_DRV\_GetRandomData() function gets random data from the RNGA module.

### Seed RNGA

1. RNGA\_DRV\_Seed() function inputs an entropy value that the RNGA can use to seed the pseudo random algorithm.

### RNGA interrupt

If the RNGA interrupt is enabled, the RNGA asserts an interrupt when an underflow error happens.

1. Enable the RNGA interrupt with the `rnga_user_config_t.isIntMasked = false`.
2. Define the RNGA IRQ function.

```
void RNGA_IRQHandler()
{
 /* Enter RNGA ISR */
}
```

### 19.2.1 Data Structure Documentation

#### 19.2.1.1 struct `rnga_user_config_t`

This structure is used when initializing the RNGA while the `rnga_init` function is called. It contains all RNGA configurations.

## Chapter 20

# Real Time Clock (RTC)

The Kinetis SDK provides both HAL and Peripheral drivers for the Real Time Clock (RTC) block of Kinetis devices.

### Modules

- [RTC HAL driver](#)

*This part describes the programming interface of the RTC HAL driver.*

- [RTC Peripheral Driver](#)

*This part describes the programming interface of the RTC Peripheral Driver.*

## RTC HAL driver

### 20.1 RTC HAL driver

This chapter describes the programming interface of the RTC HAL driver.

## Data Structures

- struct `rtc_datetime_t`  
*Structure is used to hold the time in a simple "date" format. [More...](#)*

## RTC HAL API Functions

- void `RTC_HAL_Enable` (uint32\_t rtcBaseAddr)  
*Initializes the RTC module.*
- void `RTC_HAL_Disable` (uint32\_t rtcBaseAddr)  
*Disables the RTC module.*
- void `RTC_HAL_Init` (uint32\_t rtcBaseAddr)  
*Resets the RTC module.*
- void `RTC_HAL_ConvertSecsToDatetime` (const uint32\_t \*seconds, `rtc_datetime_t` \*datetime)  
*Converts seconds to date time format data structure.*
- bool `RTC_HAL_IsDatetimeCorrectFormat` (const `rtc_datetime_t` \*datetime)  
*Checks whether the date time structure elements have the information that is within the range.*
- void `RTC_HAL_ConvertDatetimeToSecs` (const `rtc_datetime_t` \*datetime, uint32\_t \*seconds)  
*Converts the date time format data structure to seconds.*
- void `RTC_HAL_SetDatetime` (uint32\_t rtcBaseAddr, const `rtc_datetime_t` \*datetime)  
*Sets the RTC date and time according to the given time structure.*
- void `RTC_HAL_SetDatetimeInsecs` (uint32\_t rtcBaseAddr, const uint32\_t seconds)  
*Sets the RTC date and time according to the given time provided in seconds.*
- void `RTC_HAL_GetDatetime` (uint32\_t rtcBaseAddr, `rtc_datetime_t` \*datetime)  
*Gets the RTC time and stores it in the given time structure.*
- void `RTC_HAL_GetDatetimeInSecs` (uint32\_t rtcBaseAddr, uint32\_t \*seconds)  
*Gets the RTC time and returns it in seconds.*
- void `RTC_HAL_GetAlarm` (uint32\_t rtcBaseAddr, `rtc_datetime_t` \*date)  
*Reads the value of the time alarm.*
- bool `RTC_HAL_SetAlarm` (uint32\_t rtcBaseAddr, const `rtc_datetime_t` \*date)  
*Sets the RTC alarm time and enables the alarm interrupt.*

## RTC register access functions

- static uint32\_t `RTC_HAL_GetSecsReg` (uint32\_t rtcBaseAddr)  
*Reads the value of the time seconds counter.*
- static void `RTC_HAL_SetSecsReg` (uint32\_t rtcBaseAddr, const uint32\_t seconds)  
*Writes to the time seconds counter.*
- static void `RTC_HAL_SetAlarmReg` (uint32\_t rtcBaseAddr, const uint32\_t seconds)  
*Sets the time alarm and clears the time alarm flag.*
- static uint32\_t `RTC_HAL_GetAlarmReg` (uint32\_t rtcBaseAddr)  
*Gets the time alarm register contents.*
- static uint16\_t `RTC_HAL_GetPrescaler` (uint32\_t rtcBaseAddr)  
*Reads the value of the time prescaler.*

- static void [RTC\\_HAL\\_SetPrescaler](#) (uint32\_t rtcBaseAddr, const uint16\_t prescale)  
*Sets the time prescaler.*
- static uint32\_t [RTC\\_HAL\\_GetCompensationReg](#) (uint32\_t rtcBaseAddr)  
*Reads the time compensation register contents.*
- static void [RTC\\_HAL\\_SetCompensationReg](#) (uint32\_t rtcBaseAddr, const uint32\_t compValue)  
*Writes the value to the RTC TCR register.*
- static uint8\_t [RTC\\_HAL\\_GetCompensationIntervalCounter](#) (uint32\_t rtcBaseAddr)  
*Reads the current value of the compensation interval counter, which is the field CIC in the RTC TCR register.*
- static uint8\_t [RTC\\_HAL\\_GetTimeCompensationValue](#) (uint32\_t rtcBaseAddr)  
*Reads the current value used by the compensation logic for the present second interval.*
- static uint8\_t [RTC\\_HAL\\_GetCompensationIntervalRegister](#) (uint32\_t rtcBaseAddr)  
*Reads the compensation interval register.*
- static void [RTC\\_HAL\\_SetCompensationIntervalRegister](#) (uint32\_t rtcBaseAddr, const uint8\_t value)  
*Writes the compensation interval.*
- static uint8\_t [RTC\\_HAL\\_GetTimeCompensationRegister](#) (uint32\_t rtcBaseAddr)  
*Reads the time compensation value which is the configured number of 32.768 kHz clock cycles in each second.*
- static void [RTC\\_HAL\\_SetTimeCompensationRegister](#) (uint32\_t rtcBaseAddr, const uint8\_t compValue)  
*Writes to the field Time Compensation Register (TCR) of the RTC Time Compensation Register (RTC\_TC-R).*
- static void [RTC\\_HAL\\_SetOsc2pfLoadCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the oscillator configuration for the 2pF load.*
- static bool [RTC\\_HAL\\_GetOsc2pfLoad](#) (uint32\_t rtcBaseAddr)  
*Reads the oscillator 2pF load configure bit.*
- static void [RTC\\_HAL\\_SetOsc4pfLoadCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the oscillator configuration for the 4pF load.*
- static bool [RTC\\_HAL\\_GetOsc4pfLoad](#) (uint32\_t rtcBaseAddr)  
*Reads the oscillator 4pF load configure bit.*
- static void [RTC\\_HAL\\_SetOsc8pfLoadCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the oscillator configuration for the 8pF load.*
- static bool [RTC\\_HAL\\_GetOsc8pfLoad](#) (uint32\_t rtcBaseAddr)  
*Reads the oscillator 8pF load configure bit.*
- static void [RTC\\_HAL\\_SetOsc16pfLoadCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the oscillator configuration for the 16pF load.*
- static bool [RTC\\_HAL\\_GetOsc16pfLoad](#) (uint32\_t rtcBaseAddr)  
*Reads the oscillator 16pF load configure bit.*
- static void [RTC\\_HAL\\_SetClockOutCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the 32 kHz clock output to other peripherals.*
- static bool [RTC\\_HAL\\_GetClockOutCmd](#) (uint32\_t rtcBaseAddr)  
*Reads the RTC\_CR CLKO bit.*
- static void [RTC\\_HAL\\_SetOscillatorCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the oscillator.*
- static bool [RTC\\_HAL\\_IsOscillatorEnabled](#) (uint32\_t rtcBaseAddr)  
*Reads the RTC\_CR OSCE bit.*
- static void [RTC\\_HAL\\_SetUpdateModeCmd](#) (uint32\_t rtcBaseAddr, bool lock)  
*Enables/disables the update mode.*
- static bool [RTC\\_HAL\\_GetUpdateMode](#) (uint32\_t rtcBaseAddr)  
*Reads the RTC\_CR update mode bit.*

## RTC HAL driver

- static void [RTC\\_HAL\\_SetSupervisorAccessCmd](#) (uint32\_t rtcBaseAddr, bool enableRegWrite)  
*Enables/disables the supervisor access.*
- static bool [RTC\\_HAL\\_GetSupervisorAccess](#) (uint32\_t rtcBaseAddr)  
*Reads the RTC\_CR SUP bit.*
- static void [RTC\\_HAL\\_SoftwareReset](#) (uint32\_t rtcBaseAddr)  
*Performs a software reset on the RTC module.*
- static void [RTC\\_HAL\\_SoftwareResetFlagClear](#) (uint32\_t rtcBaseAddr)  
*Clears the software reset flag.*
- static bool [RTC\\_HAL\\_ReadSoftwareResetStatus](#) (uint32\_t rtcBaseAddr)  
*Reads the RTC\_CR SWR bit.*
- static bool [RTC\\_HAL\\_IsCounterEnabled](#) (uint32\_t rtcBaseAddr)  
*Reads the time counter status (enabled/disabled).*
- static void [RTC\\_HAL\\_EnableCounter](#) (uint32\_t rtcBaseAddr, bool enable)  
*Changes the time counter status.*
- static bool [RTC\\_HAL\\_HasAlarmOccured](#) (uint32\_t rtcBaseAddr)  
*Checks whether the configured time alarm has occurred.*
- static bool [RTC\\_HAL\\_HasCounterOverflowed](#) (uint32\_t rtcBaseAddr)  
*Checks whether a counter overflow has occurred.*
- static bool [RTC\\_HAL\\_IsTimeInvalid](#) (uint32\_t rtcBaseAddr)  
*Checks whether the time has been marked as invalid.*
- static void [RTC\\_HAL\\_SetLockRegistersCmd](#) (uint32\_t rtcBaseAddr, hw\_rtc\_lr\_t bitfields)  
*Configures the register lock to other module fields.*
- static bool [RTC\\_HAL\\_GetLockRegLock](#) (uint32\_t rtcBaseAddr)  
*Obtains the lock status of the lock register.*
- static void [RTC\\_HAL\\_SetLockRegLock](#) (uint32\_t rtcBaseAddr, bool lock)  
*Changes the lock status of the lock register.*
- static bool [RTC\\_HAL\\_GetStatusRegLock](#) (uint32\_t rtcBaseAddr)  
*Obtains the state of the status register lock.*
- static void [RTC\\_HAL\\_SetStatusRegLock](#) (uint32\_t rtcBaseAddr, bool lock)  
*Changes the state of the status register lock.*
- static bool [RTC\\_HAL\\_GetControlRegLock](#) (uint32\_t rtcBaseAddr)  
*Obtains the state of the control register lock.*
- static void [RTC\\_HAL\\_SetControlRegLock](#) (uint32\_t rtcBaseAddr, bool lock)  
*Changes the state of the control register lock.*
- static bool [RTC\\_HAL\\_GetTimeCompLock](#) (uint32\_t rtcBaseAddr)  
*Obtains the state of the time compensation lock.*
- static void [RTC\\_HAL\\_SetTimeCompLock](#) (uint32\_t rtcBaseAddr, bool lock)  
*Changes the state of the time compensation lock.*
- static bool [RTC\\_HAL\\_IsSecsIntEnabled](#) (uint32\_t rtcBaseAddr)  
*Checks whether the Time Seconds Interrupt is enabled/disabled.*
- static void [RTC\\_HAL\\_SetSecsIntCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the Time Seconds Interrupt.*
- static bool [RTC\\_HAL\\_ReadAlarmInt](#) (uint32\_t rtcBaseAddr)  
*Checks whether the Time Alarm Interrupt is enabled/disabled.*
- static void [RTC\\_HAL\\_SetAlarmIntCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the Time Alarm Interrupt.*
- static bool [RTC\\_HAL\\_ReadTimeOverflowInt](#) (uint32\_t rtcBaseAddr)  
*Checks whether the Time Overflow Interrupt is enabled/disabled.*
- static void [RTC\\_HAL\\_SetTimeOverflowIntCmd](#) (uint32\_t rtcBaseAddr, bool enable)  
*Enables/disables the Time Overflow Interrupt.*
- static bool [RTC\\_HAL\\_ReadTimeInvalidInt](#) (uint32\_t rtcBaseAddr)

- Checks whether the Time Invalid Interrupt is enabled/disabled.
- static void [RTC\\_HAL\\_SetTimeInvalidIntCmd](#)(uint32\_t rtcBaseAddr, bool enable)  
Enables/disables the Time Invalid Interrupt.

## 20.1.0.2 RTC HAL Driver

### Overview

The RTC HAL driver initializes the RTC registers and provides functions to read or modify the RTC registers. These are mostly invoked by the RTC Peripheral driver.

### Initialization

The HAL driver provides the enable ([RTC\\_HAL\\_Enable\(\)](#)) and disable ([RTC\\_HAL\\_Disable\(\)](#)) functions. It also provides an initialization function ([RTC\\_HAL\\_Init\(\)](#)) to reset the RTC module.

### Setting and reading the RTC time

The HAL driver provides [RTC\\_HAL\\_SetDatetime\(\)](#) function to set the date and time using an instantiation of the [rtc\\_datetime\\_t](#) structure. Details about this structure are provided here.

```
typedef struct RtcDatetime
{
 uint16_t year;
 uint16_t month;
 uint16_t day;
 uint16_t hour;
 uint16_t minute;
 uint8_t second;
} rtc_datetime_t;
```

The HAL driver also provides the ability to set the date and time in seconds using the [RTC\\_HAL\\_SetDatetimeInsecs\(\)](#) function.

### Setting and reading the Alarm

The HAL driver provides [RTC\\_HAL\\_SetAlarm\(\)](#) and [RTC\\_HAL\\_GetAlarm\(\)](#) functions to set an alarm and read back the alarm time.

## RTC HAL driver

### 20.1.1 Data Structure Documentation

#### 20.1.1.1 struct rtc\_datetime\_t

##### Data Fields

- `uint16_t year`  
*Range from 1970 to 2099.*
- `uint16_t month`  
*Range from 1 to 12.*
- `uint16_t day`  
*Range from 1 to 31 (depending on month).*
- `uint16_t hour`  
*Range from 0 to 23.*
- `uint16_t minute`  
*Range from 0 to 59.*
- `uint8_t second`  
*Range from 0 to 59.*

#### 20.1.1.0.45 Field Documentation

##### 20.1.1.0.45.1 `uint16_t rtc_datetime_t::year`

##### 20.1.1.0.45.2 `uint16_t rtc_datetime_t::month`

##### 20.1.1.0.45.3 `uint16_t rtc_datetime_t::day`

##### 20.1.1.0.45.4 `uint16_t rtc_datetime_t::hour`

##### 20.1.1.0.45.5 `uint16_t rtc_datetime_t::minute`

##### 20.1.1.0.45.6 `uint8_t rtc_datetime_t::second`

### 20.1.2 Function Documentation

#### 20.1.2.1 `void RTC_HAL_Enable ( uint32_t rtcBaseAddr )`

This function enables the RTC oscillator.

Parameters

|                          |                       |
|--------------------------|-----------------------|
| <code>rtcBaseAddr</code> | The RTC base address. |
|--------------------------|-----------------------|

#### 20.1.2.2 `void RTC_HAL_Disable ( uint32_t rtcBaseAddr )`

This function disables the RTC counter and oscillator.

Parameters

|                    |                       |
|--------------------|-----------------------|
| <i>rtcBaseAddr</i> | The RTC base address. |
|--------------------|-----------------------|

#### 20.1.2.3 void RTC\_HAL\_Init ( *uint32\_t rtcBaseAddr* )

This function initiates a soft-reset of the RTC module to reset the RTC registers.

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

#### 20.1.2.4 void RTC\_HAL\_ConvertSecsToDatetime ( *const uint32\_t \* seconds*, *rtc\_datetime\_t \* datetime* )

Parameters

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <i>seconds</i>  | holds the date and time information in seconds                       |
| <i>datetime</i> | holds the converted information from seconds in date and time format |

#### 20.1.2.5 bool RTC\_HAL\_IsDatetimeCorrectFormat ( *const rtc\_datetime\_t \* datetime* )

Parameters

|                 |                                                                           |
|-----------------|---------------------------------------------------------------------------|
| <i>datetime</i> | holds the date and time information that needs to be converted to seconds |
|-----------------|---------------------------------------------------------------------------|

#### 20.1.2.6 void RTC\_HAL\_ConvertDatetimeToSecs ( *const rtc\_datetime\_t \* datetime*, *uint32\_t \* seconds* )

Parameters

|                 |                                                                           |
|-----------------|---------------------------------------------------------------------------|
| <i>datetime</i> | holds the date and time information that needs to be converted to seconds |
| <i>seconds</i>  | holds the converted date and time in seconds                              |

## RTC HAL driver

### 20.1.2.7 void RTC\_HAL\_SetDatetime ( **uint32\_t** *rtcBaseAddr*, **const rtc\_datetime\_t \*** *datetime* )

The function converts the data from the time structure to seconds and writes the seconds value to the RTC register. The RTC counter is started after setting the time.

Parameters

|                    |                                                                              |
|--------------------|------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                         |
| <i>datetime</i>    | [in] Pointer to structure where the date and time details to set are stored. |

#### 20.1.2.8 void RTC\_HAL\_SetDatetimeInsecs ( uint32\_t *rtcBaseAddr*, const uint32\_t *seconds* )

The RTC counter is started after setting the time.

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
| <i>seconds</i>     | [in] Time in seconds |

#### 20.1.2.9 void RTC\_HAL\_GetDatetime ( uint32\_t *rtcBaseAddr*, rtc\_datetime\_t \* *datetime* )

The function reads the value in seconds from the RTC register. It then converts to the time structure which provides the time in date, hour, minutes and seconds.

Parameters

|                    |                                                                          |
|--------------------|--------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                     |
| <i>datetime</i>    | [out] pointer to a structure where the date and time details are stored. |

#### 20.1.2.10 void RTC\_HAL\_GetDatetimeInSecs ( uint32\_t *rtcBaseAddr*, uint32\_t \* *seconds* )

Parameters

|                    |                                                                   |
|--------------------|-------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                              |
| <i>datetime</i>    | [out] pointer to variable where the RTC time is stored in seconds |

#### 20.1.2.11 void RTC\_HAL\_GetAlarm ( uint32\_t *rtcBaseAddr*, rtc\_datetime\_t \* *date* )

## RTC HAL driver

Parameters

|                    |                                                                               |
|--------------------|-------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                          |
| <i>date</i>        | [out] pointer to a variable where the alarm date and time details are stored. |

### 20.1.2.12 **bool RTC\_HAL\_SetAlarm ( uint32\_t *rtcBaseAddr*, const rtc\_datetime\_t \* *date* )**

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>rtcBaseAddr</i> | The RTC base address..                                                             |
| <i>date</i>        | [in] pointer to structure where the alarm date and time details will be stored at. |

Returns

true: success in setting the RTC alarm  
false: error in setting the RTC alarm.

### 20.1.2.13 **static uint32\_t RTC\_HAL\_GetSecsReg ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

The time counter reads as zero if either the SR[TOF] or the SR[TIF] is set.

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

contents of the seconds register.

### 20.1.2.14 **static void RTC\_HAL\_SetSecsReg ( uint32\_t *rtcBaseAddr*, const uint32\_t *seconds* ) [inline], [static]**

When the time counter is enabled, the TSR is read only and increments once every second provided the S-R[TOF] or SR[TIF] is not set. When the time counter is disabled, the TSR can be read or written. Writing to the TSR when the time counter is disabled clears the SR[TOF] and/or the SR[TIF]. Writing to the TSR register with zero is supported, but not recommended, since the TSR reads as zero when either the SR[TIF] or the SR[TOF] is set (indicating the time is invalid).

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
| <i>seconds</i>     | [in] seconds value.    |

#### 20.1.2.15 static void RTC\_HAL\_SetAlarmReg ( uint32\_t *rtcBaseAddr*, const uint32\_t *seconds* ) [inline], [static]

When the time counter is enabled, the SR[TAF] is set whenever the TAR[TAR] equals the TSR[TSR] and the TSR[TSR] increments. Writing to the TAR clears the SR[TAF].

Parameters

|                    |                              |
|--------------------|------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address..       |
| <i>seconds</i>     | [in] alarm value in seconds. |

#### 20.1.2.16 static uint32\_t RTC\_HAL\_GetAlarmReg ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

contents of the alarm register.

#### 20.1.2.17 static uint16\_t RTC\_HAL\_GetPrescaler ( uint32\_t *rtcBaseAddr* ) [inline], [static]

The time counter reads as zero when either the SR[TOF] or the SR[TIF] is set.

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

contents of the time prescaler register.

## RTC HAL driver

**20.1.2.18 static void RTC\_HAL\_SetPrescaler ( uint32\_t *rtcBaseAddr*, const uint16\_t *prescale* ) [inline], [static]**

When the time counter is enabled, the TPR is read only and increments every 32.768 kHz clock cycle. When the time counter is disabled, the TPR can be read or written. The TSR[TSR] increments when bit 14 of the TPR transitions from a logic one to a logic zero.

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
| <i>prescale</i>    | Prescaler value      |

**20.1.2.19 static uint32\_t RTC\_HAL\_GetCompensationReg ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

time compensation register contents.

**20.1.2.20 static void RTC\_HAL\_SetCompensationReg ( uint32\_t *rtcBaseAddr*, const uint32\_t *compValue* ) [inline], [static]**

Parameters

|                    |                                                   |
|--------------------|---------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                              |
| <i>compValue</i>   | value to be written to the compensation register. |

**20.1.2.21 static uint8\_t RTC\_HAL\_GetCompensationIntervalCounter ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

compensation interval value.

#### 20.1.2.22 static uint8\_t RTC\_HAL\_GetTimeCompensationValue ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

time compensation value

#### 20.1.2.23 static uint8\_t RTC\_HAL\_GetCompensationIntervalRegister ( uint32\_t *rtcBaseAddr* ) [inline], [static]

The value is the configured compensation interval in seconds from 1 to 256 to control how frequently the time compensation register should adjust the number of 32.768 kHz cycles in each second. The value is one less than the number of seconds (for example, zero means a configuration for a compensation interval of one second).

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

compensation interval in seconds.

#### 20.1.2.24 static void RTC\_HAL\_SetCompensationIntervalRegister ( uint32\_t *rtcBaseAddr*, const uint8\_t *value* ) [inline], [static]

This configures the compensation interval in seconds from 1 to 256 to control how frequently the TCR should adjust the number of 32.768 kHz cycles in each second. The value written should be one less than the number of seconds (for example, write zero to configure for a compensation interval of one second). This register is double buffered and writes do not take affect until the end of the current compensation interval.

## RTC HAL driver

Parameters

|                    |                                  |
|--------------------|----------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address..           |
| <i>value</i>       | the compensation interval value. |

**20.1.2.25 static uint8\_t RTC\_HAL\_GetTimeCompensationRegister ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

time compensation value.

**20.1.2.26 static void RTC\_HAL\_SetTimeCompensationRegister ( uint32\_t *rtcBaseAddr*, const uint8\_t *compValue* ) [inline], [static]**

Configures the number of 32.768 kHz clock cycles in each second. This register is double buffered and writes do not take affect until the end of the current compensation interval. 80h Time prescaler register overflows every 32896 clock cycles. ... .

FFh Time prescaler register overflows every 32769 clock cycles.

00h Time prescaler register overflows every 32768 clock cycles.

01h Time prescaler register overflows every 32767 clock cycles.

... .

7Fh Time prescaler register overflows every 32641 clock cycles.

Parameters

|                    |                                 |
|--------------------|---------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address            |
| <i>comp_value</i>  | value of the time compensation. |

**20.1.2.27 static void RTC\_HAL\_SetOsc2pfLoadCmd ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]**

Parameters

|                    |                                                                     |
|--------------------|---------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                |
| <i>enable</i>      | can be true or false<br>true: enables load<br>false: disables load. |

#### 20.1.2.28 static bool RTC\_HAL\_GetOsc2pfLoad( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: 2pF additional load enabled.  
false: 2pF additional load disabled.

#### 20.1.2.29 static void RTC\_HAL\_SetOsc4pfLoadCmd( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]

Parameters

|                    |                                                                     |
|--------------------|---------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                |
| <i>enable</i>      | can be true or false<br>true: enables load.<br>false: disables load |

#### 20.1.2.30 static bool RTC\_HAL\_GetOsc4pfLoad( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: 4pF additional load enabled.  
false: 4pF additional load disabled.

## RTC HAL driver

**20.1.2.31 static void RTC\_HAL\_SetOsc8pfLoadCmd ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]**

Parameters

|                    |                                                                      |
|--------------------|----------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                 |
| <i>enable</i>      | can be true or false<br>true: enables load.<br>false: disables load. |

#### 20.1.2.32 static bool RTC\_HAL\_GetOsc8pfLoad ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: 8pF additional load enabled.  
false: 8pF additional load disabled.

#### 20.1.2.33 static void RTC\_HAL\_SetOsc16pfLoadCmd ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]

Parameters

|                    |                                                                      |
|--------------------|----------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                 |
| <i>enable</i>      | can be true or false<br>true: enables load.<br>false: disables load. |

#### 20.1.2.34 static bool RTC\_HAL\_GetOsc16pfLoad ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: 16pF additional load enabled.  
false: 16pF additional load disabled.

## RTC HAL driver

20.1.2.35 **static void RTC\_HAL\_SetClockOutCmd ( uint32\_t *rtcBaseAddr*, bool *enable* )**  
[**inline**], [**static**]

Parameters

|                    |                                                                                |
|--------------------|--------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                           |
| <i>enable</i>      | can be true or false<br>true: enables clock out.<br>false: disables clock out. |

#### 20.1.2.36 static bool RTC\_HAL\_GetClockOutCmd ( *uint32\_t rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: 32 kHz clock is not output to other peripherals.  
false: 32 kHz clock is output to other peripherals.

#### 20.1.2.37 static void RTC\_HAL\_SetOscillatorCmd ( *uint32\_t rtcBaseAddr, bool enable* ) [inline], [static]

After enabling, waits for the oscillator startup time before enabling the time counter to allow the 32.768 kHz clock time to stabilize.

Parameters

|                    |                                                                                  |
|--------------------|----------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                             |
| <i>enable</i>      | can be true or false<br>true: enables oscillator.<br>false: disables oscillator. |

#### 20.1.2.38 static bool RTC\_HAL\_IsOscillatorEnabled ( *uint32\_t rtcBaseAddr* ) [inline], [static]

## RTC HAL driver

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: 32.768 kHz oscillator is enabled false: 32.768 kHz oscillator is disabled.

### 20.1.2.39 static void RTC\_HAL\_SetUpdateModeCmd ( uint32\_t *rtcBaseAddr*, bool *lock* ) [inline], [static]

This mode allows the time counter enable bit in the SR to be written even when the status register is locked. When set, the time counter enable, can always be written if the TIF (Time Invalid Flag) or TOF (Time Overflow Flag) are set or if the time counter enable is clear. For devices with the monotonic counter it allows the monotonic enable to be written when it is locked. When set, the monotonic enable can always be written if the TIF (Time Invalid Flag) or TOF (Time Overflow Flag) are set or if the monotonic counter enable is clear. For devices with tamper detect it allows the it to be written when it is locked. When set, the tamper detect can always be written if the TIF (Time Invalid Flag) is clear. Note: Tamper and Monotonic features are not available in all MCUs.

Parameters

|                    |                                                                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                          |
| <i>lock</i>        | can be true or false<br>true: registers can be written when locked under limited conditions<br>false: registers cannot be written when locked |

### 20.1.2.40 static bool RTC\_HAL\_GetUpdateMode ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Registers can be written when locked under limited conditions. false: Registers cannot be written when locked.

#### **20.1.2.41 static void RTC\_HAL\_SetSupervisorAccessCmd ( uint32\_t *rtcBaseAddr*, bool *enableRegWrite* ) [inline], [static]**

This configures non-supervisor mode write access to all RTC registers and non-supervisor mode read access to RTC tamper/monotonic registers. Note: Tamper and Monotonic features are NOT available in all MCUs.

Parameters

|                             |                                                                                                                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i>          | The RTC base address..                                                                                                                                                   |
| <i>enableReg-<br/>Write</i> | can be true or false<br>true: non-supervisor mode write accesses are supported.<br>false: non-supervisor mode write accesses are not supported and generate a bus error. |

#### **20.1.2.42 static bool RTC\_HAL\_GetSupervisorAccess ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Non-supervisor mode write accesses are supported  
false: Non-supervisor mode write accesses are not supported.

#### **20.1.2.43 static void RTC\_HAL\_SoftwareReset ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

This resets all RTC registers except for the SWR bit and the RTC\_WAR and RTC\_RAR registers. The SWR bit is cleared after VBAT POR and by software explicitly clearing it. Note: access control features (RTC\_WAR and RTC\_RAR registers) are not available in all MCUs.

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

#### **20.1.2.44 static void RTC\_HAL\_SoftwareResetFlagClear ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

## RTC HAL driver

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

**20.1.2.45 static bool RTC\_HAL\_ReadSoftwareResetStatus ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: SWR is set. false: SWR is cleared.

**20.1.2.46 static bool RTC\_HAL\_IsCounterEnabled ( uint32\_t *rtcBaseAddr* ) [inline], [static]**

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: time counter is enabled, time seconds register and time prescaler register are not writeable, but increment.

false: time counter is disabled, time seconds register and time prescaler register are writeable, but do not increment.

**20.1.2.47 static void RTC\_HAL\_EnableCounter ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]**

Parameters

|                    |                                                                                             |
|--------------------|---------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                        |
| <i>enable</i>      | can be true or false<br>true: enables the time counter<br>false: disables the time counter. |

**20.1.2.48 static bool RTC\_HAL\_HasAlarmOccured ( uint32\_t *rtcBaseAddr* ) [inline],  
[static]**

Reads time alarm flag (TAF). This flag is set when the time alarm register (TAR) equals the time seconds register (TSR) and the TSR increments. This flag is cleared by writing the TAR register.

## RTC HAL driver

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

true: time alarm has occurred.

false: no time alarm occurred.

### 20.1.2.49 static bool RTC\_HAL\_HasCounterOverflowed ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the value of RTC Status Register (RTC\_SR), field Time Overflow Flag (TOF). This flag is set when the time counter is enabled and overflows. The TSR and TPR do not increment and read as zero when this bit is set. This flag is cleared by writing the TSR register when the time counter is disabled.

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

true: time overflow occurred and time counter is zero.

false: no time overflow occurred.

### 20.1.2.50 static bool RTC\_HAL\_IsTimeInvalid ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the value of RTC Status Register (RTC\_SR), field Time Invalid Flag (TIF). This flag is set on VBAT POR or software reset. The TSR and TPR do not increment and read as zero when this bit is set. This flag is cleared by writing the TSR register when the time counter is disabled.

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

true: time is INVALID and time counter is zero.

false: time is valid.

### 20.1.2.51 static void RTC\_HAL\_SetLockRegistersCmd ( uint32\_t *rtcBaseAddr*, hw\_RTC\_Ir\_t *bitfields* ) [inline], [static]

Parameters

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address..                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>bitfields</i>   | <p>[in] configuration flags:</p> <p>Valid bitfields:</p> <ul style="list-style-type: none"> <li>LRL: Lock Register Lock</li> <li>SRL: Status Register Lock</li> <li>CRL: Control Register Lock</li> <li>TCL: Time Compensation Lock</li> </ul> <p>For MCUs that have the Tamper Detect only:</p> <ul style="list-style-type: none"> <li>TIL: Tamper Interrupt Lock</li> <li>TTL: Tamper Trim Lock</li> <li>TDL: Tamper Detect Lock</li> <li>TEL: Tamper Enable Lock</li> <li>TTSL: Tamper Time Seconds Lock</li> </ul> <p>For MCUs that have the Monotonic Counter only:</p> <ul style="list-style-type: none"> <li>MCHL: Monotonic Counter High Lock</li> <li>MCLL: Monotonic Counter Low Lock</li> <li>MEL: Monotonic Enable Lock</li> </ul> |

#### 20.1.2.52 static bool RTC\_HAL\_GetLockRegLock( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the value of the field Lock Register Lock (LRL) of the RTC Lock Register (RTC\_LR).

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: lock register is not locked and writes complete as normal.  
false: lock register is locked and writes are ignored.

#### 20.1.2.53 static void RTC\_HAL\_SetLockRegLock( uint32\_t *rtcBaseAddr*, bool *lock* ) [inline], [static]

Writes to the field Lock Register Lock (LRL) of the RTC Lock Register (RTC\_LR). Once cleared, this can only be set by VBAT POR or software reset.

## RTC HAL driver

Parameters

|                    |                                                                                                                                                    |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                               |
| <i>lock</i>        | can be true or false<br>true: Lock register is not locked and writes complete as normal.<br>false: Lock register is locked and writes are ignored. |

### 20.1.2.54 static bool RTC\_HAL\_GetStatusRegLock ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the value of field Status Register Lock (SRL) of the RTC Lock Register (RTC\_LR), which is the field Status Register.

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Status register is not locked and writes complete as normal.  
false: Status register is locked and writes are ignored.

### 20.1.2.55 static void RTC\_HAL\_SetStatusRegLock ( uint32\_t *rtcBaseAddr*, bool *lock* ) [inline], [static]

Writes to the field Status Register Lock (SRL) of the RTC Lock Register (RTC\_LR). Once cleared, this can only be set by VBAT POR or software reset.

Parameters

|                    |                                                                                                                                                        |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                                   |
| <i>lock</i>        | can be true or false<br>true: Status register is not locked and writes complete as normal.<br>false: Status register is locked and writes are ignored. |

### 20.1.2.56 static bool RTC\_HAL\_GetControlRegLock ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the field Control Register Lock (CRL) value of the RTC Lock Register (RTC\_LR).

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Control register is not locked and writes complete as normal.  
false: Control register is locked and writes are ignored.

#### 20.1.2.57 static void RTC\_HAL\_SetControlRegLock ( uint32\_t *rtcBaseAddr*, bool *lock* ) [inline], [static]

Writes to the field Control Register Lock (CRL) of the RTC Lock Register (RTC\_LR). Once cleared, this can only be set by VBAT POR or software reset.

Parameters

|                    |                                                                                                                                                          |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                                     |
| <i>lock</i>        | can be true or false<br>true: Control register is not locked and writes complete as normal.<br>false: Control register is locked and writes are ignored. |

#### 20.1.2.58 static bool RTC\_HAL\_GetTimeCompLock ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the field Time Compensation Lock (TCL) value of the RTC Lock Register (RTC\_LR).

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Time compensation register is not locked and writes complete as normal.  
false: Time compensation register is locked and writes are ignored.

#### 20.1.2.59 static void RTC\_HAL\_SetTimeCompLock ( uint32\_t *rtcBaseAddr*, bool *lock* ) [inline], [static]

Writes to the field Time Compensation Lock (TCL) of the RTC Lock Register (RTC\_LR). Once cleared, this can only be set by VBAT POR or software reset.

## RTC HAL driver

Parameters

|                    |                                                                                                                                                                              |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                                                         |
| <i>lock</i>        | can be true or false<br>true: Time compensation register is not locked and writes complete as normal.<br>false: Time compensation register is locked and writes are ignored. |

### 20.1.2.60 static bool RTC\_HAL\_IsSecsIntEnabled ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the value of field Time Seconds Interrupt Enable (TSIE) of the RTC Interrupt Enable Register (RTC\_IER). The seconds interrupt is an edge-sensitive interrupt with a dedicated interrupt vector. It is generated once a second and requires no software overhead (there is no corresponding status flag to clear).

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Seconds interrupt is enabled.  
false: Seconds interrupt is disabled.

### 20.1.2.61 static void RTC\_HAL\_SetSecsIntCmd ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]

Writes to the field Time Seconds Interrupt Enable (TSIE) of the RTC Interrupt Enable Register (RTC\_IER). Note: The seconds interrupt is an edge-sensitive interrupt with a dedicated interrupt vector. It is generated once a second and requires no software overhead (there is no corresponding status flag to clear).

Parameters

|                    |                                                                                                      |
|--------------------|------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                 |
| <i>enable</i>      | can be true or false<br>true: Seconds interrupt is enabled.<br>false: Seconds interrupt is disabled. |

### 20.1.2.62 static bool RTC\_HAL\_ReadAlarmInt ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the field Time Alarm Interrupt Enable (TAIE) value of the RTC Interrupt Enable Register (RTC\_IER).

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Time alarm flag does generate an interrupt.  
false: Time alarm flag does not generate an interrupt.

#### 20.1.2.63 static void RTC\_HAL\_SetAlarmIntCmd ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]

Writes to the field Time Alarm Interrupt Enable (TAIE) of the RTC Interrupt Enable Register (RTC\_IER).

Parameters

|                    |                                                                                                                                     |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                |
| <i>enable</i>      | can be true or false<br>true: Time alarm flag does generate an interrupt.<br>false: Time alarm flag does not generate an interrupt. |

#### 20.1.2.64 static bool RTC\_HAL\_ReadTimeOverflowInt ( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the field Time Overflow Interrupt Enable (TOIE) of the value of the RTC Interrupt Enable Register (RTC\_IER).

Parameters

|                    |                        |
|--------------------|------------------------|
| <i>rtcBaseAddr</i> | The RTC base address.. |
|--------------------|------------------------|

Returns

true: Time overflow flag does generate an interrupt.  
false: Time overflow flag does not generate an interrupt.

#### 20.1.2.65 static void RTC\_HAL\_SetTimeOverflowIntCmd ( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]

Writes to the field Time Overflow Interrupt Enable (TOIE) of the RTC Interrupt Enable Register (RTC\_IER).

## RTC HAL driver

Parameters

|                    |                                                                                                                                           |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                      |
| <i>enable</i>      | can be true or false<br>true: Time overflow flag does generate an interrupt.<br>false: Time overflow flag does not generate an interrupt. |

### 20.1.2.66 static bool RTC\_HAL\_ReadTimeInvalidInt( uint32\_t *rtcBaseAddr* ) [inline], [static]

Reads the value of the field Time Invalid Interrupt Enable (TIIE) of the RTC Interrupt Enable Register (RTC\_IER).

Parameters

|                    |                      |
|--------------------|----------------------|
| <i>rtcBaseAddr</i> | The RTC base address |
|--------------------|----------------------|

Returns

true: Time invalid flag does generate an interrupt.  
false: Time invalid flag does not generate an interrupt.

### 20.1.2.67 static void RTC\_HAL\_SetTimeInvalidIntCmd( uint32\_t *rtcBaseAddr*, bool *enable* ) [inline], [static]

Writes to the field Time Invalid Interrupt Enable (TIIE) of the RTC Interrupt Enable Register (RTC\_IER).

Parameters

|                    |                                                                                                                                         |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <i>rtcBaseAddr</i> | The RTC base address                                                                                                                    |
| <i>enable</i>      | can be true or false<br>true: Time invalid flag does generate an interrupt.<br>false: Time invalid flag does not generate an interrupt. |

## 20.2 RTC Peripheral Driver

This chapter describes the programming interface of the RTC Peripheral Driver.

### Data Structures

- struct `rtc_repeat_alarm_state_t`  
*RTC repeated alarm information used by the RTC driver. [More...](#)*

### Functions

- bool `RTC_DRV_IsCounterEnabled` (uint32\_t instance)  
*Checks whether the RTC is enabled.*
- void `RTC_DRV_SetSecsIntCmd` (uint32\_t instance, bool secondsEnable)  
*Enables or disables the RTC seconds interrupt.*
- void `RTC_DRV_SetAlarmIntCmd` (uint32\_t instance, bool alarmEnable)  
*Enables or disables the alarm interrupt.*
- bool `RTC_DRV_GetAlarmIntCmd` (uint32\_t instance)  
*Reads the alarm interrupt.*
- bool `RTC_DRV_IsAlarmPending` (uint32\_t instance)  
*Reads the alarm status to see if the alarm has triggered.*
- void `RTC_DRV_SetTimeCompensation` (uint32\_t instance, uint32\_t compensationInterval, uint32\_t compensationTime)  
*Writes the compensation value to the RTC compensation register.*
- void `RTC_DRV_GetTimeCompensation` (uint32\_t instance, uint32\_t \*compensationInterval, uint32\_t \*compensationTime)  
*Reads the compensation value from the RTC compensation register.*
- void `RTC_DRV_AlarmIntAction` (uint32\_t instance)  
*Action to take when an RTC alarm interrupt is triggered.*
- void `RTC_DRV_SecsIntAction` (uint32\_t instance)  
*Action to take when an RTC seconds interrupt is triggered.*

### Initialize and Deinitialize

- void `RTC_DRV_Init` (uint32\_t instance)  
*Initializes the RTC module.*
- void `RTC_DRV_Deinit` (uint32\_t instance)  
*Disables the RTC module clock gate control.*

### RTC datetime set and get

- bool `RTC_DRV_SetDatetime` (uint32\_t instance, `rtc_datetime_t` \*datetime)  
*Sets the RTC date and time according to the given time structure.*
- void `RTC_DRV_GetDatetime` (uint32\_t instance, `rtc_datetime_t` \*datetime)  
*Gets the RTC time and stores it in the given time structure.*

### RTC alarm

- bool [RTC\\_DRV\\_SetAlarm](#) (uint32\_t instance, [rtc\\_datetime\\_t](#) \*alarmTime, bool enableAlarmInterrupt)  
*Sets the RTC alarm time and enables the alarm interrupt.*
- void [RTC\\_DRV\\_GetAlarm](#) (uint32\_t instance, [rtc\\_datetime\\_t](#) \*date)  
*Returns the RTC alarm time.*
- void [RTC\\_DRV\\_InitRepeatAlarm](#) (uint32\_t instance, [rtc\\_repeat\\_alarm\\_state\\_t](#) \*repeatAlarmState)  
*Initializes the RTC repeat alarm state structure.*
- bool [RTC\\_DRV\\_SetAlarmRepeat](#) (uint32\_t instance, [rtc\\_datetime\\_t](#) \*alarmTime, [rtc\\_datetime\\_t](#) \*alarmRepInterval)  
*Sets an alarm that is periodically repeated.*
- void [RTC\\_DRV\\_DeinitRepeatAlarm](#) (uint32\_t instance)  
*De-initializes the RTC repeat alarm state structure.*

### ISR Functions

- void [RTC\\_IRQHandler](#) (void)  
*Implementation of RTC Alarm handler named in startup code.*
- void [RTC\\_Seconds\\_IRQHandler](#) (void)  
*Implementation of RTC Seconds handler named in startup code.*

## 20.2.0.68 RTC Peripheral Driver

### Overview

The RTC Peripheral driver sets and gets the RTC clock, sets and reads the RTC alarm time, and receives notifications when an alarm is triggered.

### Initialization

To initialize, the user calls the [RTC\\_DRV\\_Init\(\)](#) function with the RTC instance number. Most SoCs have only one RTC instance. Therefore, the instance number is zero. The driver initialization function ungates the RTC module clock, initializes the RTC HAL layer driver, and enables the RTC interrupts.

This is an example how to create the `rtc_init()` function.

```
/* init the rtc module */
RTC_DRV_Init(0);
```

### Setting and reading the RTC time

The RTC driver uses an instantiation of the [rtc\\_datetime\\_t](#) structure either to configure or read the date & time. The user should call [RTC\\_DRV\\_SetDatetime\(\)](#) to configure the current date and time and call the

[RTC\\_DRV\\_GetDatetime\(\)](#) function to read the current date & time at a later time. This is an example to use these functions.

```
rtc_datetime_t datetimeToSet;

RTC_DRV_Init(0);

datetimeToSet.year = 2013;
datetimeToSet.month = 10;
datetimeToSet.day = 13;
datetimeToSet.hour = 18;
datetimeToSet.minute = 55;
datetimeToSet.second = 30;

/* set the datetime */
result = RTC_DRV_SetDatetime(0, &datetime);

/* get datetime */
RTC_DRV_GetDatetime(0, &datetime);
printf("Current datetime: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n",
 datetime.year, datetime.month, datetime.day,
 datetime.hour, datetime.minute, datetime.second);
```

## Triggering an Alarm

The user should call the [RTC\\_DRV\\_SetAlarm\(\)](#) function to set the alarm time and call the [RTC\\_DRV\\_-GetAlarm\(\)](#) function to read the configured alarm time. The user has to set the current time using the steps mentioned earlier prior to using the call to set the alarm time. The user can enable the option to trigger an interrupt when an alarm occurs. This is done by either calling the [RTC\\_DRV\\_SetAlarmIntCmd\(\)](#) function or through an argument to the [RTC\\_DRV\\_SetAlarm\(\)](#) function. This is an example to set an RTC alarm. This example causes an alarm interrupt to be triggered after 5 minutes.

```
rtc_datetime_t datetimeToSet;
rtc_datetime_t alarmTimeToSet;

RTC_DRV_Init(0);

datetimeToSet.year = 2013;
datetimeToSet.month = 10;
datetimeToSet.day = 13;
datetimeToSet.hour = 18;
datetimeToSet.minute = 55;
datetimeToSet.second = 30;

RTC_DRV_SetDatetime(0, &datetimeToSet);

alarmTimeToSet.year = datetimeToSet.year;
alarmTimeToSet.month = datetimeToSet.month;
alarmTimeToSet.day = datetimeToSet.day;
alarmTimeToSet.minute = datetimeToSet.minute + 5;
alarmTimeToSet.second = datetimeToSet.second;
RTC_DRV_SetAlarm(0, &alarmTimeToSet, true);
```

## Repeated alarm

To request for a repeated alarm, a configuration structure is available to configure the repeat alarm information. These are the structure details:

## RTC Peripheral Driver

```
typedef struct RtcRepeatAlarmState
{
 rtc_datetime_t alarmTime;
 rtc_datetime_t alarmRepTime;
} rtc_repeat_alarm_state_t;
```

The user should call [RTC\\_DRV\\_InitRepeatAlarm\(\)](#) function and provide the repeat alarm configuration structure to the RTC driver. The user should not free this structure as the RTC driver will store the pointer to this structure to configure the future alarm times.

This is an example to set a RTC repeat alarm. The alarm interrupt is triggered after 5 minutes and every minute after that.

```
rtc_repeat_alarm_state_t alarm_state;

rtc_datetime_t alarmTimeToSet;
rtc_datetime_t alarmReptime;
rtc_datetime_t datetimeToSet;

RTC_DRV_Init(0);
RTC_DRV_InitRepeatAlarm(0, &alarm_state);
RTC_DRV_SetDatetime(0, &datetimeToSet);

alarmTimeToSet.minute = datetimeToSet.minute + 5;

alarmReptime.year = 0;
alarmReptime.month = 0;
alarmReptime.day = 0;
alarmReptime.hour = 0;
alarmReptime.minute = 1;

if(!RTC_DRV_SetAlarmRepeat(0, &alarmTimeToSet, &alarmReptime))
{
 exit_error();
}
```

## Enable and Disable Alarm Interrupts

The user can call the [RTC\\_DRV\\_SetAlarmIntCmd\(\)](#) function to enable or disable the RTC alarm interrupt. The user can call the [RTC\\_DRV\\_GetAlarmIntCmd\(\)](#) function to get the state of the RTC alarm interrupt bit.

## Interrupt handler

The RTC driver provides an interrupt handler for the seconds and alarm interrupts. These handlers clear the status bits. When the repeated alarm is requested, the alarm interrupt handler uses the information provided in the [rtc\\_repeat\\_alarm\\_state\\_t](#) structure to schedule the next alarm.

To add more actions to the default handler, add calls to the functions inside the interrupt handlers [RTC IRQHandler\(\)](#) and [RTC\\_Seconds\\_IRQHandler\(\)](#) functions.

## 20.2.1 Data Structure Documentation

### 20.2.1.1 struct rtc\_repeat\_alarm\_state\_t

#### Data Fields

- `rtc_datetime_t alarmTime`  
*Set the RTC alarm time.*
- `rtc_datetime_t alarmRepTime`  
*Period for alarm to repeat, needs alarm interrupt be enabled.*

#### 20.2.1.1.0.46 Field Documentation

##### 20.2.1.1.0.46.1 `rtc_datetime_t rtc_repeat_alarm_state_t::alarmTime`

##### 20.2.1.1.0.46.2 `rtc_datetime_t rtc_repeat_alarm_state_t::alarmRepTime`

## 20.2.2 Function Documentation

### 20.2.2.1 `void RTC_DRV_Init( uint32_t instance )`

Enables the RTC clock and enables interrupts if requested by the user.

Parameters

|                       |                                     |
|-----------------------|-------------------------------------|
| <code>instance</code> | The RTC peripheral instance number. |
|-----------------------|-------------------------------------|

### 20.2.2.2 `void RTC_DRV_Deinit( uint32_t instance )`

Parameters

|                       |                                     |
|-----------------------|-------------------------------------|
| <code>instance</code> | The RTC peripheral instance number. |
|-----------------------|-------------------------------------|

### 20.2.2.3 `bool RTC_DRV_IsCounterEnabled( uint32_t instance )`

The function checks the TCE bit in the RTC control register.

Parameters

## RTC Peripheral Driver

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The RTC peripheral instance number. |
|-----------------|-------------------------------------|

Returns

true: The RTC counter is enabled  
false: The RTC counter is disabled

### 20.2.2.4 **bool RTC\_DRV\_SetDatetime ( uint32\_t *instance*, rtc\_datetime\_t \* *datetime* )**

The RTC counter is started after the time is set.

Parameters

|                 |                                                                              |
|-----------------|------------------------------------------------------------------------------|
| <i>instance</i> | The RTC peripheral instance number.                                          |
| <i>datetime</i> | [in] pointer to structure where the date and time details to set are stored. |

Returns

true: success in setting the time and starting the RTC  
false: failure. Error is because the datetime format is incorrect

### 20.2.2.5 **void RTC\_DRV\_GetDatetime ( uint32\_t *instance*, rtc\_datetime\_t \* *datetime* )**

Parameters

|                 |                                                                        |
|-----------------|------------------------------------------------------------------------|
| <i>instance</i> | The RTC peripheral instance number.                                    |
| <i>datetime</i> | [out] pointer to structure where the date and time details are stored. |

### 20.2.2.6 **void RTC\_DRV\_SetSecsIntCmd ( uint32\_t *instance*, bool *secondsEnable* )**

Parameters

|                      |                                                                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <i>instance</i>      | The RTC peripheral instance number.                                                                                                 |
| <i>secondsEnable</i> | Takes true or false<br>true: indicates seconds interrupt should be enabled<br>false: indicates seconds interrupt should be disabled |

**20.2.2.7 bool RTC\_DRV\_SetAlarm ( uint32\_t *instance*, rtc\_datetime\_t \* *alarmTime*, bool *enableAlarmInterrupt* )**

The function checks if the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

## RTC Peripheral Driver

Parameters

|                                   |                                                                                                                                 |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <i>instance</i>                   | The RTC peripheral instance number.                                                                                             |
| <i>alarmTime</i>                  | [in] pointer to structure where the alarm time is store.                                                                        |
| <i>enableAlarm-<br/>Interrupt</i> | Takes true or false<br>true: indicates alarm interrupt should be enabled<br>false: indicates alarm interrupt should be disabled |

Returns

true: success in setting the RTC alarm

false: error in setting the RTC alarm. Error is because the alarm datetime format is incorrect.

### 20.2.2.8 void RTC\_DRV\_SetAlarm ( uint32\_t *instance*, rtc\_datetime\_t \* *date* )

Parameters

|                 |                                                                              |
|-----------------|------------------------------------------------------------------------------|
| <i>instance</i> | The RTC peripheral instance number.                                          |
| <i>date</i>     | [out] pointer to structure where the alarm date and time details are stored. |

### 20.2.2.9 void RTC\_DRV\_InitRepeatAlarm ( uint32\_t *instance*, rtc\_repeat\_alarm\_state\_t \* *repeatAlarmState* )

The RTC driver uses this user-provided structure to store the alarm state information.

Parameters

|                               |                                                      |
|-------------------------------|------------------------------------------------------|
| <i>instance</i>               | The RTC peripheral instance number.                  |
| <i>repeatAlarm-<br/>State</i> | Pointer to structure where the alarm state is stored |

### 20.2.2.10 bool RTC\_DRV\_SetAlarmRepeat ( uint32\_t *instance*, rtc\_datetime\_t \* *alarmTime*, rtc\_datetime\_t \* *alarmReplInterval* )

Parameters

|                          |                                                        |
|--------------------------|--------------------------------------------------------|
| <i>instance</i>          | The RTC peripheral instance number.                    |
| <i>alarmTime</i>         | Pointer to structure where the alarm time is provided. |
| <i>alarmRep-Interval</i> | pointer to structure with the alarm repeat interval.   |

#### 20.2.2.11 void RTC\_DRV\_DeinitRepeatAlarm ( uint32\_t *instance* )

Parameters

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The RTC peripheral instance number. |
|-----------------|-------------------------------------|

#### 20.2.2.12 void RTC\_DRV\_SetAlarmIntCmd ( uint32\_t *instance*, bool *alarmEnable* )

Parameters

|                    |                                                                                                                                 |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <i>instance</i>    | The RTC peripheral instance number.                                                                                             |
| <i>alarmEnable</i> | Takes true or false<br>true: indicates alarm interrupt should be enabled<br>false: indicates alarm interrupt should be disabled |

#### 20.2.2.13 bool RTC\_DRV\_GetAlarmIntCmd ( uint32\_t *instance* )

Parameters

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The RTC peripheral instance number. |
|-----------------|-------------------------------------|

Returns

true: indicates alarm interrupt is enabled  
false: indicates alarm interrupt is disabled

#### 20.2.2.14 bool RTC\_DRV\_IsAlarmPending ( uint32\_t *instance* )

## RTC Peripheral Driver

Parameters

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The RTC peripheral instance number. |
|-----------------|-------------------------------------|

Returns

returns alarm status i.e. has the alarm triggered?

true: indicates alarm has occurred

false: indicates alarm has not occurred

### 20.2.2.15 void RTC\_DRV\_SetTimeCompensation ( *uint32\_t instance, uint32\_t compensationInterval, uint32\_t compensationTime* )

Parameters

|                              |                                                                                                               |
|------------------------------|---------------------------------------------------------------------------------------------------------------|
| <i>instance</i>              | The RTC peripheral instance number.                                                                           |
| <i>compensation-Interval</i> | User specified compensation interval that is written to the CIR field in RTC Time Compensation Register (TCR) |
| <i>compensation-Time</i>     | User specified compensation time that is written to the TCR field in RTC Time Compensation Register (TCR)     |

### 20.2.2.16 void RTC\_DRV\_GetTimeCompensation ( *uint32\_t instance, uint32\_t \* compensationInterval, uint32\_t \* compensationTime* )

Parameters

|                              |                                                                                                                                                  |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>instance</i>              | The RTC peripheral instance number.                                                                                                              |
| <i>compensation-Interval</i> | User specified pointer to store the compensation interval counter. This value is read from the CIC field in RTC Time Compensation Register (TCR) |
| <i>compensation-Time</i>     | User specified pointer to store the compensation time value. This value is read from the TCV field in RTC Time Compensation Register (TCR)       |

### 20.2.2.17 void RTC\_IRQHandler ( *void* )

Handles the RTC alarm interrupt and invokes any callback that is interested in the RTC alarm

### 20.2.2.18 void RTC\_Seconds\_IRQHandler ( *void* )

Handles the RTC seconds interrupt and invokes any callback that is interested in the RTC second tick.

**20.2.2.19 void RTC\_DRV\_AlarmIntAction ( uint32\_t *instance* )**

If the user wishes to receive alarms periodically, the RTC\_TAR register is updated using the repeat interval.

## RTC Peripheral Driver

Parameters

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The RTC peripheral instance number. |
|-----------------|-------------------------------------|

### 20.2.2.20 void RTC\_DRV\_SecsIntAction ( uint32\_t *instance* )

Disables the time seconds interrupt (TSIE) bit.

Parameters

|                 |                                     |
|-----------------|-------------------------------------|
| <i>instance</i> | The RTC peripheral instance number. |
|-----------------|-------------------------------------|

# Chapter 21

## Synchronous Audio Interface (SAI)

The Kinetis SDK provides both HAL and Peripheral drivers for the Synchronous Audio Interface (SAI) block of Kinetis devices.

### Modules

- [SAI HAL driver](#)

*This part describes the programming interface of the SAI HAL driver.*

- [SAI Peripheral driver](#)

*This part describes the programming interface of the SAI Peripheral driver.*

### 21.1 SAI HAL driver

This chapter describes the programming interface of the SAI HAL driver.

#### Enumerations

- enum `sai_protocol_t`  
*Define the bus type of sai.*
- enum `sai_master_slave_t` {  
  `kSaiMaster` = 0x0,  
  `kSaiSlave` = 0x1 }  
*Master or slave mode.*
- enum `sai_clk_polarity_t` {  
  `kSaiClkPolarityHigh` = 0x0,  
  `kSaiClkPolarityLow` = 0x1 }  
*Polarity of SAI clock.*
- enum `sai_clk_direction_t` {  
  `kSaiClkInternal` = 0x0,  
  `kSaiClkExternal` = 0x1 }  
*Clock generate direction.*
- enum `sai_data_order_t` {  
  `kSaiLSBFirst` = 0x0,  
  `kSaiMSBFirst` = 0x1 }  
*Data transfer polarity, means MSB first of LSB first.*
- enum `sai_sync_mode_t` {  
  `kSaiModeAsync` = 0x0,  
  `kSaiModeSync` = 0x1,  
  `kSaiModeSyncWithOtherTx` = 0x2,  
  `kSaiModeSyncWithOtherRx` = 0x3 }  
*Synchronous or asynchronous mode.*
- enum `sai_mclk_source_t` {  
  `kSaiMclkSourceSysclk` = 0x0,  
  `kSaiMclkSourceSelect1` = 0x1,  
  `kSaiMclkSourceSelect2` = 0x2,  
  `kSaiMclkSourceSelect3` = 0x3 }  
*Mater clock source.*
- enum `sai_bclk_source_t` {  
  `kSaiBclkSourceBusclk` = 0x0,  
  `kSaiBclkSourceMclkDiv` = 0x1,  
  `kSaiBclkSourceOtherSai0` = 0x2,  
  `kSaiBclkSourceOtherSai1` = 0x3 }  
*Bit clock source.*
- enum `sai_interrupt_request_t` {  
  `kSaiIntrequestWordStart` = 0x0,  
  `kSaiIntrequestSyncError` = 0x1,  
  `kSaiIntrequestFIFOWarning` = 0x2,  
  `kSaiIntrequestFIFOError` = 0x3,

- ```
kSaiIntrequestFIFORequest = 0x4 }
```

The SAI state flag.
- enum `sai_dma_request_t` {


```
kSaiDmaReqFIFOWarning = 0x0,
```

```
kSaiDmaReqFIFORequest = 0x1 }
```

The DMA request sources.
- enum `sai_state_flag_t` {


```
kSaiStateFlagWordStart = 0x0,
```

```
kSaiStateFlagSyncError = 0x1,
```

```
kSaiStateFlagFIFOError = 0x2,
```

```
kSaiStateFlagSoftReset = 0x5 }
```

The SAI state flag.
- enum `sai_reset_type_t` {


```
kSaiResetTypeSoftware = 0x0,
```

```
kSaiResetTypeFIFO = 0x1 }
```

The reset type.
- enum `sai_run_mode_t` {


```
kSaiRunModeDebug = 0x0,
```

```
kSaiRunModeStop = 0x1 }
```

Functions

- void `SAI_HAL_TxSetSyncMode` (uint32_t saiBaseAddr, `sai_sync_mode_t` sync_mode)
SAI Tx sync mode setting.
- void `SAI_HAL_RxSetSyncMode` (uint32_t saiBaseAddr, `sai_sync_mode_t` sync_mode)
SAI Rx sync mode setting.
- static uint8_t `SAI_HAL_TxGetFifoReadPointer` (uint32_t saiBaseAddr, uint32_t fifo_channel)
Gets the Tx FIFO read pointer.
- static uint8_t `SAI_HAL_RxGetFifoReadPointer` (uint32_t saiBaseAddr, uint32_t fifo_channel)
Gets the Rx FIFO read pointer.
- static uint8_t `SAI_HAL_TxGetFifoWritePointer` (uint32_t saiBaseAddr, uint32_t fifo_channel)
Gets the Tx FIFO write pointer.
- static uint8_t `SAI_HAL_RxGetFifoWritePointer` (uint32_t saiBaseAddr, uint32_t fifo_channel)
Gets the Rx FIFO write pointer.
- static uint32_t * `SAI_HAL_TxGetFifoAddr` (uint32_t saiBaseAddr, uint32_t fifo_channel)
Gets the TDR register address.
- static uint32_t * `SAI_HAL_RxGetFifoAddr` (uint32_t saiBaseAddr, uint32_t fifo_channel)
Gets the RDR register address.
- static void `SAI_HAL_TxEnable` (uint32_t saiBaseAddr)
Enables the SAI Tx module.
- static void `SAI_HAL_RxEnable` (uint32_t saiBaseAddr)
Enables the SAI Rx module.
- static void `SAI_HAL_TxDisable` (uint32_t saiBaseAddr)
Disables the Tx module.
- static void `SAI_HAL_RxDisable` (uint32_t saiBaseAddr)
Disables the Rx module.
- void `SAI_HAL_TxSetIntCmd` (uint32_t saiBaseAddr, `sai_interrupt_request_t` source, bool enable)
Enables the Tx interrupt from different interrupt sources.
- void `SAI_HAL_RxSetIntCmd` (uint32_t saiBaseAddr, `sai_interrupt_request_t` source, bool enable)

SAI HAL driver

- **bool SAI_HAL_TxGetIntCmd** (uint32_t saiBaseAddr, [sai_interrupt_request_t](#) source)
Gets the status as to whether the Tx interrupt source is enabled.
- **bool SAI_HAL_RxGetIntCmd** (uint32_t saiBaseAddr, [sai_interrupt_request_t](#) source)
Gets the status as to whether the Rx interrupt source is enabled.
- **void SAI_HAL_TxSetDmaCmd** (uint32_t saiBaseAddr, [sai_dma_request_t](#) source, bool enable)
Enables the Tx DMA request from different sources.
- **void SAI_HAL_RxSetDmaCmd** (uint32_t saiBaseAddr, [sai_dma_request_t](#) source, bool enable)
Enables the Rx DMA request from different sources.
- **bool SAI_HAL_TxGetDmaCmd** (uint32_t saiBaseAddr, [sai_dma_request_t](#) source)
Gets the status whether the Tx DMA source is enabled.
- **bool SAI_HAL_RxGetDmaCmd** (uint32_t saiBaseAddr, [sai_dma_request_t](#) source)
Gets the status whether the Rx DMA source is enabled.
- **void SAI_HAL_TxClearStateFlag** (uint32_t saiBaseAddr, [sai_state_flag_t](#) flag)
Clears the Tx state flags.
- **void SAI_HAL_RxClearStateFlag** (uint32_t saiBaseAddr, [sai_state_flag_t](#) flag)
Clears the Rx state flags.
- **void SAI_HAL_TxSetReset** (uint32_t saiBaseAddr, [sai_reset_type_t](#) type)
Resets the Tx module.
- **void SAI_HAL_RxSetReset** (uint32_t saiBaseAddr, [sai_reset_type_t](#) type)
Resets the Rx module.
- **static void SAI_HAL_TxSetWordMask** (uint32_t saiBaseAddr, uint32_t mask)
Sets the Tx mask word of the frame.
- **static void SAI_HAL_RxSetWordMask** (uint32_t saiBaseAddr, uint32_t mask)
Sets the Rx mask word of the frame.
- **static void SAI_HAL_TxSetDataChn** (uint32_t saiBaseAddr, uint8_t fifo_channel)
Sets the Tx FIFO channel.
- **static void SAI_HAL_RxSetDataChn** (uint32_t saiBaseAddr, uint8_t fifo_channel)
Sets the Rx FIFO channel.
- **void SAI_HAL_TxSetRunModeCmd** (uint32_t saiBaseAddr, [sai_run_mode_t](#) run_mode, bool enable)
Sets the running mode of the Tx.
- **void SAI_HAL_RxSetRunModeCmd** (uint32_t saiBaseAddr, [sai_run_mode_t](#) run_mode, bool enable)
Sets the running mode of the Rx.
- **static void SAI_HAL_TxSetWordStartIndex** (uint32_t saiBaseAddr, uint32_t index)
Configures at which word the start of word flag is set in the Tx.
- **static void SAI_HAL_RxSetWordStartIndex** (uint32_t saiBaseAddr, uint32_t index)
Configures at which word the start of word flag is set in the Rx.
- **bool SAI_HAL_TxGetStateFlag** (uint32_t saiBaseAddr, [sai_state_flag_t](#) flag)
Gets the state of the flags in the TCSR.
- **bool SAI_HAL_RxGetStateFlag** (uint32_t saiBaseAddr, [sai_state_flag_t](#) flag)
Gets the state of the flags in the RCSR.
- **static uint32_t SAI_HAL_ReceiveData** (uint32_t saiBaseAddr, uint32_t rx_channel)
Receives the data from the FIFO.
- **static void SAI_HAL_SendData** (uint32_t saiBaseAddr, uint32_t tx_channel, uint32_t data)
Transmits data to the FIFO.
- **uint32_t SAI_HAL_ReceiveDataBlocking** (uint32_t saiBaseAddr, uint32_t rx_channel)
Uses blocking to receive data.
- **void SAI_HAL_SendDataBlocking** (uint32_t saiBaseAddr, uint32_t tx_channel, uint32_t data)
Uses blocking to send data.

Module control

- void **SAI_HAL_TxInit** (uint32_t saiBaseAddr)
Initializes the SAI Tx.
- void **SAI_HAL_RxInit** (uint32_t saiBaseAddr)
Initializes the SAI Rx.
- void **SAI_HAL_TxSetProtocol** (uint32_t saiBaseAddr, **sai_protocol_t** protocol)
Sets Tx protocol relevant settings.
- void **SAI_HAL_RxSetProtocol** (uint32_t saiBaseAddr, **sai_protocol_t** protocol)
Sets Rx protocol relevant settings.
- void **SAI_HAL_TxSetMasterSlave** (uint32_t saiBaseAddr, **sai_master_slave_t** master_slave_mode)
Sets master or slave mode.
- void **SAI_HAL_RxSetMasterSlave** (uint32_t saiBaseAddr, **sai_master_slave_t** master_slave_mode)
Sets master or slave mode.

Master clock configuration

- static void **SAI_HAL_SetMclkSrc** (uint32_t saiBaseAddr, **sai_mclk_source_t** source)
Sets the master clock source.
- static uint32_t **SAI_HAL_GetMclkSrc** (uint32_t saiBaseAddr)
Gets the master clock source.
- static void **SAI_HAL_SetMclkDividerCmd** (uint32_t saiBaseAddr, bool enable)
Sets the direction of the SAI master clock.
- void **SAI_HAL_SetMclkDiv** (uint32_t saiBaseAddr, uint32_t mclk, uint32_t src_clk)
Sets the divider of the master clock.
- static bool **SAI_HAL_GetMclkDivUpdatingCmd** (uint32_t saiBaseAddr)
Flag to see if the master clock divider is re-divided.

Bit clock configuration

- static void **SAI_HAL_TxSetBclkSrc** (uint32_t saiBaseAddr, **sai_bclk_source_t** source)
Sets the bit clock source of Tx.
- static void **SAI_HAL_RxSetBclkSrc** (uint32_t saiBaseAddr, **sai_bclk_source_t** source)
Sets bit clock source of the Rx.
- static uint32_t **SAI_HAL_TxGetBclkSrc** (uint32_t saiBaseAddr)
Gets the bit clock source of Tx.
- static uint32_t **SAI_HAL_RxGetBclkSrc** (uint32_t saiBaseAddr)
Gets bit clock source of the Rx.
- static void **SAI_HAL_TxSetBclkDiv** (uint32_t saiBaseAddr, uint32_t divider)
Sets the Tx bit clock divider value.
- static void **SAI_HAL_RxSetBclkDiv** (uint32_t saiBaseAddr, uint32_t divider)
Sets the Rx bit clock divider value.
- static void **SAI_HAL_TxSetBclkCmd** (uint32_t saiBaseAddr, bool enable)
Enables or disables the Tx bit clock.
- static void **SAI_HAL_RxSetBclkCmd** (uint32_t saiBaseAddr, bool enable)
Enables or disables the Rx bit clock.
- static void **SAI_HAL_TxSetBclkInputCmd** (uint32_t saiBaseAddr, bool enable)
Enables or disables the Tx bit clock input bit.

SAI HAL driver

- static void **SAI_HAL_RxSetBclkInputCmd** (uint32_t saiBaseAddr, bool enable)
Enables or disables the Rx bit clock input bit.
- static void **SAI_HAL_TxSetSwapBclkCmd** (uint32_t saiBaseAddr, bool enable)
Sets the Tx bit clock swap.
- static void **SAI_HAL_RxSetSwapBclkCmd** (uint32_t saiBaseAddr, bool enable)
Sets the Rx bit clock swap.
- static void **SAI_HAL_TxSetBclkDir** (uint32_t saiBaseAddr, **sai_clk_direction_t** direction)
Sets the direction of the Tx SAI bit clock.
- static void **SAI_HAL_RxSetBclkDir** (uint32_t saiBaseAddr, **sai_clk_direction_t** direction)
Sets the direction of the Rx SAI bit clock.
- static void **SAI_HAL_TxSetBclkPolarity** (uint32_t saiBaseAddr, **sai_clk_polarity_t** pol)
Sets the polarity of the Tx SAI bit clock.
- static void **SAI_HAL_RxSetBclkPolarity** (uint32_t saiBaseAddr, **sai_clk_polarity_t** pol)
Sets the polarity of the Rx SAI bit clock.

Frame sync configuration

- static void **SAI_HAL_TxSetFrameSize** (uint32_t saiBaseAddr, uint32_t size)
Sets the Tx frame size.
- static void **SAI_HAL_RxSetFrameSize** (uint32_t saiBaseAddr, uint32_t size)
Sets the Rx frame size.
- static uint32_t **SAI_HAL_TxGetFrameSize** (uint32_t saiBaseAddr)
Gets the Tx frame size.
- static uint32_t **SAI_HAL_RxGetFrameSize** (uint32_t saiBaseAddr)
Gets the Rx frame size.
- static void **SAI_HAL_TxSetFrameSyncWidth** (uint32_t saiBaseAddr, uint32_t width)
Sets the Tx sync width.
- static void **SAI_HAL_RxSetFrameSyncWidth** (uint32_t saiBaseAddr, uint32_t width)
Sets the Rx sync width.
- static void **SAI_HAL_TxSetFrameSyncPolarity** (uint32_t saiBaseAddr, **sai_clk_polarity_t** pol)
Sets the polarity of the Tx frame sync.
- static void **SAI_HAL_RxSetFrameSyncPolarity** (uint32_t saiBaseAddr, **sai_clk_polarity_t** pol)
Sets the polarity of the Rx frame sync.
- static void **SAI_HAL_TxSetFrameSyncDir** (uint32_t saiBaseAddr, **sai_clk_direction_t** direction)
Sets the direction of the SAI Tx frame sync.
- static void **SAI_HAL_RxSetFrameSyncDir** (uint32_t saiBaseAddr, **sai_clk_direction_t** direction)
Sets the direction of the SAI Rx frame sync.
- static void **SAI_HAL_TxSetBitOrder** (uint32_t saiBaseAddr, **sai_data_order_t** order)
Sets the Tx data transfer order.
- static void **SAI_HAL_RxSetBitOrder** (uint32_t saiBaseAddr, **sai_data_order_t** order)
Sets the Rx data transfer order.
- static void **SAI_HAL_TxSetFrameSyncEarlyCmd** (uint32_t saiBaseAddr, bool enable)
Tx Frame sync one bit early.
- static void **SAI_HAL_RxSetFrameSyncEarlyCmd** (uint32_t saiBaseAddr, bool enable)
Rx Frame sync one bit early.

Word configurations

- static void **SAI_HAL_TxSetWordSize** (uint32_t saiBaseAddr, uint32_t bits)

- static void **SAI_HAL_RxSetWordSize** (uint32_t saiBaseAddr, uint32_t bits)

Sets the word size for Rx.
- static uint32_t **SAI_HAL_TxGetWordSize** (uint32_t saiBaseAddr)

Gets the Tx word size.
- static uint32_t **SAI_HAL_RxGetWordSize** (uint32_t saiBaseAddr)

Gets the Rx word size.
- static void **SAI_HAL_TxSetFirstWordSize** (uint32_t saiBaseAddr, uint8_t size)

Sets the size of the first word of the Tx frame .
- static void **SAI_HAL_RxSetFirstWordSize** (uint32_t saiBaseAddr, uint8_t size)

Sets the size of the first word of Rx frame .
- static void **SAI_HAL_TxSetFirstBitShifted** (uint32_t saiBaseAddr, uint32_t index)

Sets the FIFO index for the first bit data.
- static void **SAI_HAL_RxSetFirstBitShifted** (uint32_t saiBaseAddr, uint32_t index)

Sets the index in FIFO for the first bit data.

watermark settings

- static void **SAI_HAL_TxSetWatermark** (uint32_t saiBaseAddr, uint32_t watermark)

Sets the Tx watermark value.
- static void **SAI_HAL_RxSetWatermark** (uint32_t saiBaseAddr, uint32_t watermark)

Sets the Rx watermark value.
- static uint32_t **SAI_HAL_TxGetWatermark** (uint32_t saiBaseAddr)

Gets the Tx watermark value.
- static uint32_t **SAI_HAL_RxGetWatermark** (uint32_t saiBaseAddr)

Gets the Rx watermark value.

21.1.0.21 SAI HAL Driver

Overview

The SAI HAL driver masks the hardware and provide a comprehensible way to use SAI.

21.1.1 Enumeration Type Documentation

21.1.1.1 enum sai_master_slave_t

Enumerator

kSaiMaster Master mode.

kSaiSlave Slave mode.

SAI HAL driver

21.1.1.2 enum sai_clk_polarity_t

Enumerator

kSaiClkPolarityHigh Clock active high.

kSaiClkPolarityLow Clock active low.

21.1.1.3 enum sai_clk_direction_t

Enumerator

kSaiClkInternal Clock generated internal.

kSaiClkExternal Clock generated external.

21.1.1.4 enum sai_data_order_t

Enumerator

kSaiLSBFFirst Least significant bit transferred first.

kSaiMSBFFirst Most significant bit transferred first.

21.1.1.5 enum sai_sync_mode_t

Enumerator

kSaiModeAsync Asynchronous mode.

kSaiModeSync Synchronous mode (with receiver or transmit)

kSaiModeSyncWithOtherTx Synchronous with another SAI transmit.

kSaiModeSyncWithOtherRx Synchronous with another SAI receiver.

21.1.1.6 enum sai_mclk_source_t

Enumerator

kSaiMclkSourceSysclk Master clock from the system clock.

kSaiMclkSourceSelect1 Master clock from source 1.

kSaiMclkSourceSelect2 Master clock from source 2.

kSaiMclkSourceSelect3 Master clock from source 3.

21.1.1.7 enum sai_bclk_source_t

Enumerator

- kSaiBclkSourceBusclk* Bit clock using bus clock.
- kSaiBclkSourceMclkDiv* Bit clock using master clock divider.
- kSaiBclkSourceOtherSai0* Bit clock from other SAI device.
- kSaiBclkSourceOtherSai1* Bit clock from other SAI device.

21.1.1.8 enum sai_interrupt_request_t

Enumerator

- kSaiIntrequestWordStart* Word start flag, means the first word in a frame detected.
- kSaiIntrequestSyncError* Sync error flag, means the sync error is detected.
- kSaiIntrequestFIFOWarning* FIFO warning flag, means the FIFO is empty.
- kSaiIntrequestFIFOError* FIFO error flag.
- kSaiIntrequestFIFORequest* FIFO request, means reached watermark.

21.1.1.9 enum sai_dma_request_t

Enumerator

- kSaiDmaReqFIFOWarning* FIFO warning caused by the DMA request.
- kSaiDmaReqFIFORequest* FIFO request caused by the DMA request.

21.1.1.10 enum sai_state_flag_t

Enumerator

- kSaiStateFlagWordStart* Word start flag, means the first word in a frame detected.
- kSaiStateFlagSyncError* Sync error flag, means the sync error is detected.
- kSaiStateFlagFIFOError* FIFO error flag.
- kSaiStateFlagSoftReset* Software reset flag.

21.1.1.11 enum sai_reset_type_t

Enumerator

- kSaiResetTypeSoftware* Software reset, reset the logic state.
- kSaiResetTypeFIFO* FIFO reset, reset the FIFO read and write pointer.

SAI HAL driver

21.1.1.12 enum sai_run_mode_t

Enumerator

kSaiRunModeDebug In debug mode.

kSaiRunModeStop In stop mode.

21.1.2 Function Documentation

21.1.2.1 void SAI_HAL_TxInit (uint32_t *saiBaseAddr*)

The initialization resets the SAI module by setting the SR bit of TCSR register. Note that the function writes 0 to every control registers.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.2 void SAI_HAL_RxInit (uint32_t *saiBaseAddr*)

The initialization resets the SAI module by setting the SR bit of RCSR register. Note that the function writes 0 to every control registers.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.3 void SAI_HAL_TxSetProtocol (uint32_t *saiBaseAddr*, sai_protocol_t *protocol*)

The bus mode means which protocol SAI uses. It can be I2S left, right and so on. Each protocol has a different configuration on bit clock and frame sync.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>protocol</i>	The protocol selection. It can be I2S left aligned, I2S right aligned, etc.

21.1.2.4 void SAI_HAL_RxSetProtocol (uint32_t *saiBaseAddr*, sai_protocol_t *protocol*)

The bus mode means which protocol SAI uses. It can be I2S left, right and so on. Each protocol has a different configuration on bit clock and frame sync.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>protocol</i>	The protocol selection. It can be I2S left aligned, I2S right aligned, etc.

21.1.2.5 void SAI_HAL_TxSetMasterSlave (uint32_t *saiBaseAddr*, sai_master_slave_t *master_slave_mode*)

The function determines master or slave mode. Master mode provides its own clock and slave mode uses an external clock.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>master_slave_mode</i>	Mater or slave mode.

21.1.2.6 void SAI_HAL_RxSetMasterSlave (uint32_t *saiBaseAddr*, sai_master_slave_t *master_slave_mode*)

The function determines master or slave mode. Master mode provides its own clock and slave mode uses external clock.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>master_slave_mode</i>	Mater or slave mode.

21.1.2.7 static void SAI_HAL_SetMclkSrc (uint32_t *saiBaseAddr*, sai_mclk_source_t *source*) [inline], [static]

The source of the clock is different from socs. This function sets the clock source for SAI master clock source. Master clock is used to produce the bit clock for the data transfer.

Parameters

SAI HAL driver

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	Mater clock source

21.1.2.8 static uint32_t SAI_HAL_GetMclkSrc (uint32_t *saiBaseAddr*) [inline], [static]

The source of the clock is different from socs. This function gets the clock source for SAI master clock source. Master clock is used to produce the bit clock for the data transfer.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

Returns

Mater clock source

21.1.2.9 static void SAI_HAL_SetMclkDividerCmd (uint32_t *saiBaseAddr*, bool *enable*) [inline], [static]

This function would decides the direction of bit clock generated.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means enable, false means disable.

21.1.2.10 void SAI_HAL_SetMclkDiv (uint32_t *saiBaseAddr*, uint32_t *mclk*, uint32_t *src_clk*)

Using the divider to get the master clock frequency wanted from the source. $mclk = clk_source * fract/divide$. The input is the master clock frequency needed and the source clock frequency. The master clock is decided by the sample rate and the multi-clock number. Notice that mclk should less than src_clk, or it would do hang as the HW refuses to write in this situation.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>mclk</i>	Master clock frequency needed.
<i>src_clk</i>	The source clock frequency.

21.1.2.11 **static bool SAI_HAL_GetMclkDivUpdatingCmd (uint32_t *saiBaseAddr*)**
[**inline**], [**static**]

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

Returns

True if the divider updated otherwise false.

21.1.2.12 static void SAI_HAL_TxSetBclkSrc (uint32_t *saiBaseAddr*, sai_bclk_source_t *source*) [inline], [static]

It is generated by the master clock, bus clock and other devices.

The function sets the source of the bit clock. The bit clock can be produced by the master clock and from the bus clock or other SAI Tx/Rx. Tx and Rx in the SAI module use the same bit clock either from Tx or Rx.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	Bit clock source.

21.1.2.13 static void SAI_HAL_RxSetBclkSrc (uint32_t *saiBaseAddr*, sai_bclk_source_t *source*) [inline], [static]

It is generated by the master clock, bus clock and other devices.

The function sets the source of the bit clock. The bit clock can be produced by the master clock, and from the bus clock or other SAI Tx/Rx. Tx and Rx in the SAI module use the same bit clock either from Tx or Rx.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	Bit clock source.

21.1.2.14 static uint32_t SAI_HAL_TxGetBclkSrc (uint32_t *saiBaseAddr*) [inline], [static]

It is generated by the master clock, bus clock and other devices.

The function gets the source of the bit clock. The bit clock can be produced by the master clock and from the bus clock or other SAI Tx/Rx. Tx and Rx in the SAI module use the same bit clock either from Tx or Rx.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

Returns

Bit clock source.

21.1.2.15 static uint32_t SAI_HAL_RxGetBclkSrc (uint32_t *saiBaseAddr*) [inline], [static]

It is generated by the master clock, bus clock and other devices.

The function gets the source of the bit clock. The bit clock can be produced by the master clock, and from the bus clock or other SAI Tx/Rx. Tx and Rx in the SAI module use the same bit clock either from Tx or Rx.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

Returns

Bit clock source.

21.1.2.16 static void SAI_HAL_TxSetBclkDiv (uint32_t *saiBaseAddr*, uint32_t *divider*) [inline], [static]

bclk = mclk / divider. At the same time, bclk = sample_rate * channel * bits. This means how much time is needed to transfer one bit. Notice: The function is called while the bit clock source is the master clock.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>divider</i>	The divide number of bit clock.

21.1.2.17 static void SAI_HAL_RxSetBclkDiv (uint32_t *saiBaseAddr*, uint32_t *divider*) [inline], [static]

bclk = mclk / divider. At the same time, bclk = sample_rate * channel * bits. This means how much time is needed to transfer one bit. Notice: The function is called while the bit clock source is the master clock.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>divider</i>	The divide number of bit clock.

**21.1.2.18 static void SAI_HAL_TxSetBclkCmd (uint32_t *saiBaseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means enable, false means disable.

**21.1.2.19 static void SAI_HAL_RxSetBclkCmd (uint32_t *saiBaseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means enable, false means disable.

**21.1.2.20 static void SAI_HAL_TxSetBclkInputCmd (uint32_t *saiBaseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means enable, false means disable.

**21.1.2.21 static void SAI_HAL_RxSetBclkInputCmd (uint32_t *saiBaseAddr*, bool *enable*)
[inline], [static]**

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means enable, false means disable.

21.1.2.22 static void SAI_HAL_TxSetSwapBclkCmd (uint32_t *saiBaseAddr*, bool *enable*) [inline], [static]

This field swaps the bit clock used by the transmitter. When the transmitter is configured in asynchronous mode and this bit is set, the transmitter is clocked by the receiver bit clock. This allows the transmitter and receiver to share the same bit clock, but the transmitter continues to use the transmit frame sync (S-AI_TX_SYNC). When the transmitter is configured in synchronous mode, the transmitter BCS field and receiver BCS field must be set to the same value. When both are set, the transmitter and receiver are both clocked by the transmitter bit clock (SAI_TX_BCLK) but use the receiver frame sync (SAI_RX_SYNC).

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means swap bit closk, false means no swap.

21.1.2.23 static void SAI_HAL_RxSetSwapBclkCmd (uint32_t *saiBaseAddr*, bool *enable*) [inline], [static]

This field swaps the bit clock used by the receiver. When the receiver is configured in asynchronous mode and this bit is set, the receiver is clocked by the transmitter bit clock (SAI_RX_BCLK). This allows the transmitter and receiver to share the same bit clock, but the receiver continues to use the receiver frame sync (SAI_RX_SYNC). When the receiver is configured in synchronous mode, the transmitter BCS field and receiver BCS field must be set to the same value. When both are set, the transmitter and receiver are both clocked by the receiver bit clock (SAI_RX_BCLK) but use the transmitter frame sync (SAI_TX_SYNC).

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means swap bit closk, false means no swap.

21.1.2.24 static void SAI_HAL_TxSetBclkDir (uint32_t *saiBaseAddr*, sai_clk_direction_t *direction*) [inline], [static]

This function sets the direction of the bit clock generated.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>direction</i>	Bit clock generated internal or external.

21.1.2.25 static void SAI_HAL_RxSetBclkDir(uint32_t *saiBaseAddr*, sai_clk_direction_t *direction*) [inline], [static]

This function sets the direction of the bit clock generated.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>direction</i>	Bit clock generated internal or external.

21.1.2.26 static void SAI_HAL_TxSetBclkPolarity(uint32_t *saiBaseAddr*, sai_clk_polarity_t *pol*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>pol</i>	Polarity of the SAI bit clock, which can be configured to active high or low.

21.1.2.27 static void SAI_HAL_RxSetBclkPolarity(uint32_t *saiBaseAddr*, sai_clk_polarity_t *pol*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>pol</i>	Polarity of SAI bit clock, which can be configured to active high or low.

21.1.2.28 static void SAI_HAL_TxSetFrameSize(uint32_t *saiBaseAddr*, uint32_t *size*) [inline], [static]

The frame size means how many words are in a frame. For example 2-channel audio data, the frame size is 2, which means 2 words in a frame.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>size</i>	Words number in a frame.

21.1.2.29 static void SAI_HAL_RxSetFrameSize (uint32_t *saiBaseAddr*, uint32_t *size*) [inline], [static]

The frame size means how many words are in a frame. For example 2-channel audio data, the frame size is 2, which means 2 words in a frame.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>size</i>	Words number in a frame.

21.1.2.30 static uint32_t SAI_HAL_TxGetFrameSize (uint32_t *saiBaseAddr*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.31 static uint32_t SAI_HAL_RxGetFrameSize (uint32_t *saiBaseAddr*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.32 static void SAI_HAL_TxSetFrameSyncWidth (uint32_t *saiBaseAddr*, uint32_t *width*) [inline], [static]

A sync is the number of bit clocks of a frame. The sync width cannot be longer than the length of the first word of the frame.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>width</i>	How many bit clock in a sync.

21.1.2.33 static void SAI_HAL_RxSetFrameSyncWidth (uint32_t *saiBaseAddr*, uint32_t *width*) [inline], [static]

A sync is the number of bit clocks of a frame. The sync width cannot be longer than the length of the first word of the frame.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>width</i>	How many bit clock in a sync.

21.1.2.34 static void SAI_HAL_TxSetFrameSyncPolarity (uint32_t *saiBaseAddr*, sai_clk_polarity_t *pol*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>pol</i>	Polarity of sai frame sync, can be configured to active high or low.

21.1.2.35 static void SAI_HAL_RxSetFrameSyncPolarity (uint32_t *saiBaseAddr*, sai_clk_polarity_t *pol*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module..
<i>pol</i>	Polarity of SAI frame sync, can be configured to active high or low.

21.1.2.36 static void SAI_HAL_TxSetFrameSyncDir (uint32_t *saiBaseAddr*, sai_clk_direction_t *direction*) [inline], [static]

This function sets the direction of frame sync.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>direction</i>	Frame sync generated internal or external.

21.1.2.37 static void SAI_HAL_RxSetFrameSyncDir (uint32_t *saiBaseAddr*, sai_clk_direction_t *direction*) [inline], [static]

This function sets the direction of frame sync.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>direction</i>	Frame sync generated internal or external.

21.1.2.38 static void SAI_HAL_TxSetBitOrder (uint32_t *saiBaseAddr*, sai_data_order_t *order*) [inline], [static]

This function sets the data transfer order. It can be set to MSB first or LSB first.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>order</i>	MSB transmit first or LSB transmit first.

21.1.2.39 static void SAI_HAL_RxSetBitOrder (uint32_t *saiBaseAddr*, sai_data_order_t *order*) [inline], [static]

This function sets the data transfer order. It can be set to MSB first or LSB first.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>order</i>	MSB transmit first or LSB transmit first.

21.1.2.40 static void SAI_HAL_TxSetFrameSyncEarlyCmd (uint32_t *saiBaseAddr*, bool *enable*) [inline], [static]

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means the frame sync one bit early and false means no early.

21.1.2.41 static void SAI_HAL_RxSetFrameSyncEarlyCmd (uint32_t *saiBaseAddr*, bool *enable*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>enable</i>	True means the frame sync one bit early and false means no early.

21.1.2.42 static void SAI_HAL_TxSetWordSize (uint32_t *saiBaseAddr*, uint32_t *bits*) [inline], [static]

The word size means the quantization level of audio file. SAI supports the 8 bit, 16 bit, 24 bit, and 32 bit formats.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>bits</i>	How many bits in a word.

21.1.2.43 static void SAI_HAL_RxSetWordSize (uint32_t *saiBaseAddr*, uint32_t *bits*) [inline], [static]

The word size means the quantization level of audio file. SAI supports 8 bit, 16 bit, 24 bit, and 32 bit formats.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>bits</i>	How many bits in a word.

21.1.2.44 static uint32_t SAI_HAL_TxGetWordSize (uint32_t *saiBaseAddr*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.45 static uint32_t SAI_HAL_RxGetWordSize (uint32_t *saiBaseAddr*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.46 static void SAI_HAL_TxSetFirstWordSize (uint32_t *saiBaseAddr*, uint8_t *size*) [inline], [static]

In I2S protocol, the size of the first word is the same as the size of other words. In some protocols, for example, AC'97, the first word is not the same size as others. This function sets the length of the first word which is, in most situations, the same as others.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>size</i>	The length of frame head word.

21.1.2.47 static void SAI_HAL_RxSetFirstWordSize (uint32_t *saiBaseAddr*, uint8_t *size*) [inline], [static]

In I2S protocol, the size of the first word is the same as the size of other words. In some protocols, for example, AC'97, the first word is not the same size as others. This function sets the length of the first word which is, in most situations, the same as others.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>size</i>	The length of frame head word.

21.1.2.48 static void SAI_HAL_TxSetFirstBitShifted (uint32_t *saiBaseAddr*, uint32_t *index*) [inline], [static]

The FIFO is 32-bit in SAI. However, not all audio data is 32-bit, but is mostly 16-bit. In this situation, the codec needs to know which bit of the FIFO marks the valid audio data.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>index</i>	First bit shifted in FIFO.

21.1.2.49 static void SAI_HAL_RxSetFirstBitShifted (uint32_t *saiBaseAddr*, uint32_t *index*) [inline], [static]

The FIFO is 32-bit in SAI. However, not all audio data is 32-bit, but is mostly 16-bit. In this situation, the codec needs to know which bit of the FIFO marks the valid audio data.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>index</i>	First bit shifted in FIFO.

21.1.2.50 static void SAI_HAL_TxSetWatermark (uint32_t *saiBaseAddr*, uint32_t *watermark*) [inline], [static]

While the value in the FIFO is less or equal to the watermark , it generates an interrupt request or a DMA request. The watermark value cannot be greater than the depth of FIFO.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>watermark</i>	Watermark value of a FIFO.

21.1.2.51 static void SAI_HAL_RxSetWatermark (uint32_t *saiBaseAddr*, uint32_t *watermark*) [inline], [static]

While the value in the FIFO is more or equal to the watermark , it generates an interrupt request or a DMA request. The watermark value cannot be greater than the depth of FIFO.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

<i>watermark</i>	Watermark value of a FIFO.
------------------	----------------------------

**21.1.2.52 static uint32_t SAI_HAL_TxGetWatermark (uint32_t *saiBaseAddr*)
[inline], [static]**

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

**21.1.2.53 static uint32_t SAI_HAL_RxGetWatermark (uint32_t *saiBaseAddr*)
[inline], [static]**

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

**21.1.2.54 void SAI_HAL_TxSetSyncMode (uint32_t *saiBaseAddr*, sai_sync_mode_t
sync_mode)**

The mode can be asynchronous mode, synchronous, or synchronous with another SAI device. When configured for a synchronous mode of operation, the receiver must be configured for the asynchronous operation.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>sync_mode</i>	Synchronous mode or Asynchronous mode.

**21.1.2.55 void SAI_HAL_RxSetSyncMode (uint32_t *saiBaseAddr*, sai_sync_mode_t
sync_mode)**

The mode can be asynchronous mode, synchronous, or synchronous with another SAI device. When configured for a synchronous mode of operation, the receiver must be configured for the asynchronous operation.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>sync_mode</i>	Synchronous mode or Asynchronous mode.

21.1.2.56 static uint8_t SAI_HAL_TxGetFifoReadPointer (uint32_t *saiBaseAddr*, uint32_t *fifo_channel*) [inline], [static]

It is used to determine whether the FIFO is full or empty and know how much space there is for FIFO. If *read_ptr* == *write_ptr*, the FIFO is empty. While the bit of the *read_ptr* and the *write_ptr* are equal except for the MSB, the FIFO is full.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel selected.

Returns

FIFO read pointer value.

21.1.2.57 static uint8_t SAI_HAL_RxGetFifoReadPointer (uint32_t *saiBaseAddr*, uint32_t *fifo_channel*) [inline], [static]

It is used to determine whether the FIFO is full or empty and know how much space there is for FIFO. If *read_ptr* == *write_ptr*, the FIFO is empty. While the bit of the *read_ptr* and the *write_ptr* are equal except for the MSB, the FIFO is full.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel selected.

Returns

FIFO read pointer value.

21.1.2.58 static uint8_t SAI_HAL_TxGetFifoWritePointer (uint32_t *saiBaseAddr*, uint32_t *fifo_channel*) [inline], [static]

It is used to determine whether the FIFO is full or empty and know how much space there is for FIFO. If *read_ptr* == *write_ptr*, the FIFO is empty. While the bit of the *read_ptr* and *write_ptr* are equal except for the MSB, the FIFO is full.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel selected.

Returns

FIFO read pointer value.

21.1.2.59 static uint8_t SAI_HAL_RxGetFifoWritePointer (uint32_t *saiBaseAddr*, uint32_t *fifo_channel*) [inline], [static]

It is used to determine whether the FIFO is full or empty and know how much space there is for FIFO. If *read_ptr* == *write_ptr*, the FIFO is empty. While the bit of the *read_ptr* and *write_ptr* are equal except for the MSB, the FIFO is full.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel selected.

Returns

FIFO read pointer value.

21.1.2.60 static uint32_t* SAI_HAL_TxGetFifoAddr (uint32_t *saiBaseAddr*, uint32_t *fifo_channel*) [inline], [static]

This function determines the dest/src address of the DMA transfer.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel selected.

Returns

TDR register or RDR register address

21.1.2.61 static uint32_t* SAI_HAL_RxGetFifoAddr (uint32_t *saiBaseAddr*, uint32_t *fifo_channel*) [inline], [static]

This function determines the dest/src address of the DMA transfer.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel selected.

Returns

TDR register or RDR register address

21.1.2.62 static void SAI_HAL_TxEnable (uint32_t *saiBaseAddr*) [inline], [static]

Enables the Tx. This function enables both the bit clock and the transfer channel.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.63 static void SAI_HAL_RxEnable (uint32_t *saiBaseAddr*) [inline], [static]

Enables the Rx. This function enables both the bit clock and the receive channel.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.64 static void SAI_HAL_TxDisable (uint32_t *saiBaseAddr*) [inline], [static]

Disables the Tx. This function disables both the bit clock and the transfer channel.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.65 static void SAI_HAL_RxDisable (uint32_t *saiBaseAddr*) [inline], [static]

Disables the Rx. This function disables both the bit clock and the receive channel.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
--------------------	--------------------------------------

21.1.2.66 void SAI_HAL_TxSetIntCmd (uint32_t *saiBaseAddr*, sai_interrupt_request_t *source*, bool *enable*)

The interrupt source can be : Word start flag, Sync error flag, FIFO error flag, FIFO warning flag, FIFO request flag. This function sets which flag causes an interrupt request.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI interrupt request source.
<i>enable</i>	Enable or disable.

21.1.2.67 void SAI_HAL_RxSetIntCmd (uint32_t *saiBaseAddr*, sai_interrupt_request_t *source*, bool *enable*)

The interrupt source can be : Word start flag, Sync error flag, FIFO error flag, FIFO warning flag, FIFO request flag. This function sets which flag causes an interrupt request.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI interrupt request source.
<i>enable</i>	Enable or disable.

21.1.2.68 bool SAI_HAL_TxGetIntCmd (uint32_t *saiBaseAddr*, sai_interrupt_request_t *source*)

The interrupt source can be : Word start flag, Sync error flag, FIFO error flag, FIFO warning flag, FIFO request flag. This function sets which flag causes an interrupt request.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI interrupt request source.

SAI HAL driver

Returns

Enabled or disabled.

21.1.2.69 **bool SAI_HAL_RxGetIntCmd (uint32_t *saiBaseAddr*, sai_interrupt_request_t *source*)**

The interrupt source can be : Word start flag, Sync error flag, FIFO error flag, FIFO warning flag, FIFO request flag. This function sets which flag causes an interrupt request.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI interrupt request source.

Returns

Enabled or disabled.

21.1.2.70 **void SAI_HAL_TxSetDmaCmd (uint32_t *saiBaseAddr*, sai_dma_request_t *source*, bool *enable*)**

The DMA sources can be: FIFO warning and FIFO request. This function enables the DMA request from different DMA request sources.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI DMA request source.
<i>enable</i>	Enable or disable.

21.1.2.71 **void SAI_HAL_RxSetDmaCmd (uint32_t *saiBaseAddr*, sai_dma_request_t *source*, bool *enable*)**

The DMA sources can be: FIFO warning and FIFO request. This function enables the DMA request from different DMA request sources.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI DMA request source.
<i>enable</i>	Enable or disable.

21.1.2.72 bool SAI_HAL_TxGetDmaCmd (uint32_t *saiBaseAddr*, sai_dma_request_t *source*)

The DMA sources can be: FIFO warning and FIFO request. This function enables the DMA request from different DMA request sources.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI DMA request source.
<i>Enable</i>	or disable.

21.1.2.73 bool SAI_HAL_RxGetDmaCmd (uint32_t *saiBaseAddr*, sai_dma_request_t *source*)

The DMA sources can be: FIFO warning and FIFO request. This function enables the DMA request from different DMA request sources.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>source</i>	SAI DMA request source.

Returns

Enable or disable.

21.1.2.74 void SAI_HAL_TxClearStateFlag (uint32_t *saiBaseAddr*, sai_state_flag_t *flag*)

The function is used to clear the flags manually. It can clear word start, FIFO warning, FIFO error, FIFO request flag.

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>flag</i>	SAI state flag type. The flag can be word start, sync error, FIFO error/warning.

21.1.2.75 void SAI_HAL_RxClearStateFlag (uint32_t *saiBaseAddr*, sai_state_flag_t *flag*)

The function is used to clear the flags manually. It can clear word start, FIFO warning, FIFO error, FIFO request flag.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>flag</i>	SAI state flag type. The flag can be word start, sync error, FIFO error/warning.

21.1.2.76 void SAI_HAL_TxSetReset (uint32_t *saiBaseAddr*, sai_reset_type_t *type*)

There are two kinds of resets: Software reset and FIFO reset. Software reset: resets all transmitter internal logic, including the bit clock generation, status flags and FIFO pointers. It does not reset the configuration registers. FIFO reset: synchronizes the FIFO write pointer to the same value as the FIFO read pointer. This empties the FIFO contents and is to be used after the Transmit FIFO Error Flag is set, and before the FIFO is re-initialized and the Error Flag is cleared.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>type</i>	SAI reset type.

21.1.2.77 void SAI_HAL_RxSetReset (uint32_t *saiBaseAddr*, sai_reset_type_t *type*)

There are two kinds of resets: Software reset and FIFO reset. Software reset: resets all transmitter internal logic, including the bit clock generation, status flags and FIFO pointers. It does not reset the configuration registers. FIFO reset: synchronizes the FIFO write pointer to the same value as the FIFO read pointer. This empties the FIFO contents and is to be used after the Transmit FIFO Error Flag is set, and before the FIFO is re-initialized and the Error Flag is cleared.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>type</i>	SAI reset type.

21.1.2.78 static void SAI_HAL_TxSetWordMask (uint32_t *saiBaseAddr*, uint32_t *mask*) [inline], [static]

Each bit number represent the mask word index. For example, 0 represents mask the 0th word, 3 represents mask 0th and 1st word. The TMR register can be different from frame to frame. If the user wants a mono audio, set the mask to 0/1.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>mask</i>	Which bits need to be masked in a frame.

21.1.2.79 static void SAI_HAL_RxSetWordMask (uint32_t *saiBaseAddr*, uint32_t *mask*) [inline], [static]

Each bit number represent the mask word index. For example, 0 represents mask the 0th word, 3 represents mask 0th and 1st word. The TMR register can be different from frame to frame. If the user wants a mono audio, set the mask to 0/1.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>mask</i>	Which bits need to be masked in a frame.

21.1.2.80 static void SAI_HAL_TxSetDataChn (uint32_t *saiBaseAddr*, uint8_t *fifo_channel*) [inline], [static]

A SAI *saiBaseAddr* includes a Tx and an Rx. Each has several channels according to different platforms. A channel means a path for the audio data input/output.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel number.

SAI HAL driver

21.1.2.81 static void SAI_HAL_RxSetDataChn (uint32_t *saiBaseAddr*, uint8_t *fifo_channel*) [inline], [static]

A SAI *saiBaseAddr* includes a Tx and a Rx. Each has several channels according to different platforms. A channel means a path for the audio data input/output.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>fifo_channel</i>	FIFO channel number.

21.1.2.82 void SAI_HAL_TxSetRunModeCmd (uint32_t *saiBaseAddr*, sai_run_mode_t *run_mode*, bool *enable*)

There is a debug mode, stop mode, and a normal mode.

This function can set the working mode of the SAI *saiBaseAddr*. Stop mode is always used in low power cases, and the debug mode disables the SAI after the current transmit/receive is completed.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>run_mode</i>	SAI running mode.
<i>enable</i>	Enable or disable a mode.

21.1.2.83 void SAI_HAL_RxSetRunModeCmd (uint32_t *saiBaseAddr*, sai_run_mode_t *run_mode*, bool *enable*)

There is a debug mode, stop mode, and a normal mode.

This function can set the working mode of the SAI *saiBaseAddr*. Stop mode is always used in low power cases, and the debug mode disables the SAI after the current transmit/receive is completed.

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>run_mode</i>	SAI running mode.
<i>enable</i>	Enable or disable a mode.

21.1.2.84 static void SAI_HAL_TxSetWordStartIndex (uint32_t *saiBaseAddr*, uint32_t *index*) [inline], [static]

SAI HAL driver

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>index</i>	Which word triggers word start flag.

21.1.2.85 static void SAI_HAL_RxSetWordStartIndex (uint32_t *saiBaseAddr*, uint32_t *index*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>index</i>	Which word triggers word start flag.

21.1.2.86 bool SAI_HAL_TxGetStateFlag (uint32_t *saiBaseAddr*, sai_state_flag_t *flag*)

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>flag</i>	State flag type, it can be FIFO error, FIFO warning and so on.

Returns

True if detect word start otherwise false.

21.1.2.87 bool SAI_HAL_RxGetStateFlag (uint32_t *saiBaseAddr*, sai_state_flag_t *flag*)

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>flag</i>	State flag type, it can be FIFO error, FIFO warning and so on.

Returns

True if detect word start otherwise false.

21.1.2.88 static uint32_t SAI_HAL_ReceiveData (uint32_t *saiBaseAddr*, uint32_t *rx_channel*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>rx_channel</i>	Rx FIFO channel.
<i>data</i>	Pointer to the address to be written in.

21.1.2.89 static void SAI_HAL_SendData (uint32_t *saiBaseAddr*, uint32_t *tx_channel*, uint32_t *data*) [inline], [static]

Parameters

<i>saiBaseAddr</i>	Register base address of SAI module.
<i>tx_channel</i>	Tx FIFO channel.
<i>data</i>	Data value which needs to be written into FIFO.

21.1.2.90 uint32_t SAI_HAL_ReceiveDataBlocking (uint32_t *saiBaseAddr*, uint32_t *rx_channel*)

Parameters

<i>saiBaseAddr</i>	The SAI saiBaseAddr.
<i>rx_channel</i>	Rx FIFO channel.

Returns

Received data.

21.1.2.91 void SAI_HAL_SendDataBlocking (uint32_t *saiBaseAddr*, uint32_t *tx_channel*, uint32_t *data*)

Parameters

<i>saiBaseAddr</i>	The SAI saiBaseAddr.
<i>tx_channel</i>	Tx FIFO channel.
<i>data</i>	Data value which needs to be written into FIFO.

21.2 SAI Peripheral driver

This chapter describes the programming interface of the SAI Peripheral driver.

Data Structures

- struct [sai_data_format_t](#)
Defines the PCM data format. [More...](#)
- struct [sai_state_t](#)
SAI internal state. Users should allocate and transfer memory to the PD during the initialization function. [More...](#)
- struct [sai_user_config_t](#)
The description structure for the SAI TX/RX module. [More...](#)

Typedefs

- [typedef void\(* sai_callback_t \)](#)(void *parameter)
SAI callback function.

Enumerations

- enum [sai_status_t](#)
Status structure for SAI.

Functions

- [sai_status_t SAI_DRV_TxInit](#) (uint32_t instance, [sai_user_config_t](#) *config, [sai_state_t](#) *state)
Initializes the SAI module.
- [sai_status_t SAI_DRV_RxInit](#) (uint32_t instance, [sai_user_config_t](#) *config, [sai_state_t](#) *state)
Initializes the SAI Rx module.
- [void SAI_DRV_TxGetDefaultSetting](#) ([sai_user_config_t](#) *config)
This function gets the default setting of the user configuration.
- [void SAI_DRV_RxGetDefaultSetting](#) ([sai_user_config_t](#) *config)
This function gets the default setting of the user configuration.
- [sai_status_t SAI_DRV_TxDeinit](#) (uint32_t instance)
De-initializes the SAI Tx module.
- [sai_status_t SAI_DRV_RxDeinit](#) (uint32_t instance)
De-initializes the SAI Rx module.
- [sai_status_t SAI_DRV_TxConfigDataFormat](#) (uint32_t instance, [sai_data_format_t](#) *format)
Configures audio data format of Tx.
- [sai_status_t SAI_DRV_RxConfigDataFormat](#) (uint32_t instance, [sai_data_format_t](#) *format)
Configures audio data format of Rx.
- [void SAI_DRV_TxStartModule](#) (uint32_t instance)
Starts the Tx transfer.
- [void SAI_DRV_RxStartModule](#) (uint32_t instance)
Starts the Rx receive process.

- static void **SAI_DRV_TxStopModule** (uint32_t instance)

Stops writing data to FIFO to disable the DMA or the interrupt request bit.
- static void **SAI_DRV_RxStopModule** (uint32_t instance)

Stops receiving data from FIFO to disable the DMA or the interrupt request bit.
- static void **SAI_DRV_TxSetIntCmd** (uint32_t instance, bool enable)

Enables or disables the Tx interrupt source.
- static void **SAI_DRV_RxSetIntCmd** (uint32_t instance, bool enable)

Enables or disables the Rx interrupt source.
- static void **SAI_DRV_TxSetDmaCmd** (uint32_t instance, bool enable)

Enables or disables the Tx DMA source.
- static void **SAI_DRV_RxSetDmaCmd** (uint32_t instance, bool enable)

Enables or disables the Rx interrupt source.
- void **SAI_DRV_TxSetWatermark** (uint32_t instance, uint32_t watermark)

Sets the Tx watermark.
- void **SAI_DRV_RxSetWatermark** (uint32_t instance, uint32_t watermark)

Sets the Rx watermark.
- static uint32_t **SAI_DRV_TxGetWatermark** (uint32_t instance)

Gets the Tx watermark.
- static uint32_t **SAI_DRV_RxGetWatermark** (uint32_t instance)

Gets the Rx watermark.
- static uint32_t * **SAI_DRV_TxGetFifoAddr** (uint32_t instance, uint32_t fifo_channel)

Gets the Tx FIFO address of the data channel.
- static uint32_t * **SAI_DRV_RxGetFifoAddr** (uint32_t instance, uint32_t fifo_channel)

Gets the Rx FIFO address of the data channel.
- uint32_t **SAI_DRV_SendData** (uint32_t instance, uint8_t *addr, uint32_t len)

Sends date of a certain length.
- uint32_t **SAI_DRV_ReceiveData** (uint32_t instance, uint8_t *addr, uint32_t len)

Receives a certain length data.
- uint32_t **SAI_DRV_SendDataBlocking** (uint32_t instance, uint8_t *addr, uint32_t len)

Sends a certain length data in blocking way.
- uint32_t **SAI_DRV_ReceiveDataBlocking** (uint32_t instance, uint8_t *addr, uint32_t len)

Receives data of a certain length in blocking way.
- void **SAI_DRV_TxRegisterCallback** (uint32_t instance, **sai_callback_t** callback, void *callback_param)

Registers the callback function after completing a send.
- void **SAI_DRV_RxRegisterCallback** (uint32_t instance, **sai_callback_t** callback, void *callback_param)

Registers the callback function after completing a receive.
- void **SAI_DRV_TxIRQHandler** (uint32_t instance)

Default SAI Tx interrupt handler.
- void **SAI_DRV_RxIRQHandler** (uint32_t instance)

Default SAI Rx interrupt handler.

21.2.0.92 SAI Driver

Overview

The SAI driver initializes, configures, starts, and stops the SAI. The SAI driver also implements configuration functions, sends and receives data.

SAI Peripheral driver

Initialization

To initialize SAI, call the [SAI_DRV_TxInit\(\)](#) or [SAI_DRV_RxInit\(\)](#) function and pass the parameters needed. The parameter is the SAI instance and SAI configuration structure. The function opens the clock gate and initializes the SAI modules according to the structure information.

Configuration

Configuration is implemented by the [SAI_DRV_TxConfigDataFormat\(\)](#) function. To use this function, transfer the audio data format to the SAI module.

Call diagram

To use the SAI driver, follow these steps, and use Tx as an example:

1. Initialize the SAI module by calling the [SAI_DRV_TxInit\(\)](#) function.
2. Configure the audio data features of SAI by calling the [SAI_DRV_TxConfigDataFormat\(\)](#) function.
3. Send/receive data by calling the [SAI_DRV_SendData\(\)](#) or the [SAI_DRV_ReceiveData\(\)](#) functions.
4. Start Tx or Rx by calling the [SAI_DRV_TxStartModule\(\)](#) or the [SAI_DRV_RxStartModule\(\)](#) function.
5. Shut down the SAI module by calling the [SAI_DRV_TxDeinit\(\)](#) function.

This is example code to initialize and configure the SAI driver in the DMA mode:

```
// Initialize handler structure.
sai_handler_t handler;
handler.direction = AUDIO_TX;
handler.instance = 0;
handler.fifo_channel = 0;

//Initialize config structure.
sai_user_config_t tx_config;
tx_config.bus_type = kSaiBusI2SLeft;
tx_config.channel = 0;
tx_config.slave_master = kSaiMaster;
tx_config.sync_mode = kSaiModeAsync;
tx_config.bclk_source = kSaiBclkSourceMclkDiv;
tx_config.mclk_source = kSaiMclkSourceSysclk;
tx_config.mclk_divide_enable = true;
tx_config.watermark = 4;

//SAI state, used for driver internal logic.
sai_state_t tx_state;

//Data format of audio data
audio_data_format_t format;
format.bits = 16;
format.sample_rate = 44100;
format.mclk = 384 * format->sample_rate;
format.words = 2;
//Initialize SAI Tx.
SAI_DRV_TxInit(instance, &tx_config, &tx_state);
//Configure the data format of SAI tx.
SAI_DRV_TxConfigDataFormat(instance,&format);
```

```

//option: register callback functions for finished transfer
SAI_DRV_TxRegisterCallback(instance, callback, callback_param);
//start send data
SAI_DRV_SendData(instance, addr, len);
//Enable interrupt
SAI_DRV_TxIntCmd(instance, true);
//Start Tx
SAI_DRV_TxStartModule(instance);

.....
//Stop Tx.
SAI_DRV_TxStopModule(instance);
//Deinit Tx.
SAI_DRV_TxDeinit(instance);

```

21.2.1 Data Structure Documentation

21.2.1.1 struct sai_data_format_t

Data Fields

- **uint32_t sample_rate**
Sample rate of the PCM file.
- **uint32_t mclk**
Master clock frequency.
- **uint8_t bits**
How many bits in a word.
- **sai_mono_streo_t mono_streo**
How many word in a frame.

21.2.1.2 struct sai_state_t

Note: During the SAI execution, users should not free the state. Otherwise, the driver malfunctions.

21.2.1.3 struct sai_user_config_t

Data Fields

- **sai_mclk_source_t mclk_source**
Master clock source.
- **bool mclk_divide_enable**
Enable the divide of master clock to generate bit clock.
- **sai_sync_mode_t sync_mode**
Synchronous or asynchronous.
- **sai_protocol_t protocol**
I2S left, I2S right or I2S type.
- **sai_master_slave_t slave_master**
Master or slave.
- **sai_bclk_source_t bclk_source**

SAI Peripheral driver

- *Bit clock from master clock or other modules.*
• `uint8_t channel`
Which FIFO is used to transfer.

21.2.1.3.0.47 Field Documentation

21.2.1.3.0.47.1 `sai_mclk_source_t sai_user_config_t::mclk_source`

21.2.1.3.0.47.2 `bool sai_user_config_t::mclk_divide_enable`

21.2.1.3.0.47.3 `sai_sync_mode_t sai_user_config_t::sync_mode`

21.2.1.3.0.47.4 `sai_protocol_t sai_user_config_t::protocol`

21.2.1.3.0.47.5 `sai_master_slave_t sai_user_config_t::slave_master`

21.2.1.3.0.47.6 `sai_bclk_source_t sai_user_config_t::bclk_source`

21.2.1.3.0.47.7 `uint8_t sai_user_config_t::channel`

21.2.2 Function Documentation

21.2.2.1 `sai_status_t SAI_DRV_TxInit (uint32_t instance, sai_user_config_t * config, sai_state_t * state)`

This function initializes the SAI registers according to the configuration structure. This function also initializes the basic SAI settings including board-relevant settings. Notice: This function does not initialize an entire SAI instance. It only initializes the TX according to the value in the handler.

Parameters

<code>instance</code>	SAI module instance.
<code>config</code>	The configuration structure of SAI.
<code>state</code>	Pointer of SAI run state structure.

Returns

Return kStatus_SAI_Success while the initialize success and kStatus_SAI_Fail if failed.

21.2.2.2 `sai_status_t SAI_DRV_RxInit (uint32_t instance, sai_user_config_t * config, sai_state_t * state)`

This function initializes the SAI registers according to the configuration structure. This function also initializes the basic SAI settings including board-relevant settings. Note that this function does not initialize an entire SAI instance. This function only initializes the TX according to the value in the handler.

Parameters

<i>instance</i>	SAI module instance.
<i>config</i>	The configuration structure of SAI.
<i>state</i>	Pointer of SAI run state structure.

Returns

Return kStatus_SAI_Success while the initialize success and kStatus_SAI_Fail if failed.

21.2.2.3 void SAI_DRV_TxGetDefaultSetting (*sai_user_config_t * config*)

The default settings for SAI are:

- Audio protocol is I2S format
- Watermark is 4
- Use SAI0
- Channel is channel0
- SAI as master
- MCLK from system core clock
- Tx is in an asynchronous mode

Parameters

<i>config</i>	Pointer of user configure structure.
---------------	--------------------------------------

21.2.2.4 void SAI_DRV_RxGetDefaultSetting (*sai_user_config_t * config*)

The default settings for SAI are:

- Audio protocol is I2S format
- Watermark is 4
- Use SAI0
- Data channel is channel0
- SAI as master
- MCLK from system core clock
- Rx is in synchronous way

Parameters

SAI Peripheral driver

<i>config</i>	Pointer of user configure structure.
---------------	--------------------------------------

21.2.2.5 **sai_status_t SAI_DRV_TxDeinit (uint32_t *instance*)**

This function closes the SAI Tx device. However, it does not close the entire SAI instance. It only closes the clock gate while both Tx and Rx are closed in the same instance.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

Returns

Return kStatus_SAI_Success while the process success and kStatus_SAI_Fail if failed.

21.2.2.6 **sai_status_t SAI_DRV_RxDeinit (uint32_t *instance*)**

This function closes the SAI Rx device. However, it does not close the entire SAI instance. It only closes the clock gate while both Tx and Rx are closed in the same instance.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

Returns

Return kStatus_SAI_Success while the process success and kStatus_SAI_Fail if failed.

21.2.2.7 **sai_status_t SAI_DRV_TxConfigDataFormat (uint32_t *instance*, sai_data_format_t * *format*)**

The function configures an audio sample rate, data bits, and a channel number.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

<i>format</i>	PCM data format structure pointer.
---------------	------------------------------------

Returns

Return kStatus_SAI_Success while the process success and kStatus_SAI_Fail if failed.

21.2.2.8 **sai_status_t SAI_DRV_RxConfigDataFormat (uint32_t *instance*, sai_data_format_t * *format*)**

The function configures an audio sample rate, data bits, and a channel number.

Parameters

<i>instance</i>	SAI module instance of the SAI module.
<i>format</i>	PCM data format structure pointer.

Returns

Return kStatus_SAI_Success while the process success and kStatus_SAI_Fail if failed.

21.2.2.9 **void SAI_DRV_TxStartModule (uint32_t *instance*)**

The function enables the interrupt/DMA request source and the transmit channel.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

21.2.2.10 **void SAI_DRV_RxStartModule (uint32_t *instance*)**

The function enables the interrupt/DMA request source and the transmit channel.

Parameters

<i>instance</i>	SAI module instance of the SAI module.
-----------------	--

21.2.2.11 **static void SAI_DRV_TxStopModule (uint32_t *instance*) [inline], [static]**

This function provides the method to pause writing data.

SAI Peripheral driver

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

21.2.2.12 static void SAI_DRV_RxStopModule (uint32_t *instance*) [inline], [static]

This function provides the method to pause writing data.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

21.2.2.13 static void SAI_DRV_TxSetIntCmd (uint32_t *instance*, bool *enable*) [inline], [static]

Parameters

<i>instance</i>	SAI module instance.
<i>enable</i>	True means enable interrupt source, false means disable interrupt source.

21.2.2.14 static void SAI_DRV_RxSetIntCmd (uint32_t *instance*, bool *enable*) [inline], [static]

Parameters

<i>instance</i>	SAI module instance.
<i>enable</i>	True means enable interrupt source, false means disable interrupt source.

21.2.2.15 static void SAI_DRV_TxSetDmaCmd (uint32_t *instance*, bool *enable*) [inline], [static]

Parameters

<i>instance</i>	SAI module instance.
<i>enable</i>	True means enable DMA source, false means disable DMA source.

21.2.2.16 **static void SAI_DRV_RxSetDmaCmd (uint32_t *instance*, bool *enable*)**
[**inline**], [**static**]

SAI Peripheral driver

Parameters

<i>instance</i>	SAI module instance.
<i>enable</i>	True means enable DMA source, false means disable DMA source.

21.2.2.17 void SAI_DRV_TxSetWatermark (uint32_t *instance*, uint32_t *watermark*)

Setting the watermark means that while the data number in FIFO is less or equal to the watermark, it generates an interrupt request or the DMA request.

Parameters

<i>instance</i>	SAI module instance.
<i>watermark</i>	Watermark number needs to set.

21.2.2.18 void SAI_DRV_RxSetWatermark (uint32_t *instance*, uint32_t *watermark*)

Setting the watermark means that while the data number in FIFO is more or equal to the watermark, it generates an interrupt request or the DMA request.

Parameters

<i>instance</i>	SAI module instance.
<i>watermark</i>	Watermark number needs to set.

21.2.2.19 static uint32_t SAI_DRV_TxGetWatermark (uint32_t *instance*) [inline], [static]

The watermark should be changed according to a different audio sample rate.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

Returns

Watermark number in TCR1.

**21.2.2.20 static uint32_t SAI_DRV_RxGetWatermark (uint32_t *instance*) [inline],
[static]**

The watermark should be changed according to a different audio sample rate.

SAI Peripheral driver

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

Returns

Watermark number in RCR1.

21.2.2.21 static uint32_t* SAI_DRV_TxGetFifoAddr (uint32_t *instance*, uint32_t *fifo_channel*) [inline], [static]

This function is mainly used for the DMA settings, which the DMA configuration needs for the source/destination address of SAI.

Parameters

<i>instance</i>	SAI module instance of the SAI module.
<i>fifo_channel</i>	FIFO channel of SAI Tx.

Returns

Returns the address of the data channel FIFO.

21.2.2.22 static uint32_t* SAI_DRV_RxGetFifoAddr (uint32_t *instance*, uint32_t *fifo_channel*) [inline], [static]

This function is mainly used for the DMA settings, which the DMA configuration needs for the source/destination address of SAI.

Parameters

<i>instance</i>	SAI module instance of the SAI module.
<i>fifo_channel</i>	FIFO channel of SAI Rx.

Returns

Returns the address of the data channel FIFO.

21.2.2.23 uint32_t SAI_DRV_SendData (uint32_t *instance*, uint8_t * *addr*, uint32_t *len*)

This function sends the data to the Tx FIFO. This function starts the transfer, and, while finishing the transfer, calls the callback function registered by users.

Parameters

<i>instance</i>	SAI module instance of the SAI module.
<i>addr</i>	Address of the data which needs to be transferred.
<i>len</i>	The data length which need to be sent.

Returns

Returns the length which was sent.

21.2.2.24 **uint32_t SAI_DRV_ReceiveData (uint32_t *instance*, uint8_t * *addr*, uint32_t *len*)**

This function receives the data from the RX FIFO. This function starts the transfer, and, while finishing the transfer, calls the callback function registered by users.

Parameters

<i>instance</i>	SAI module instance.
<i>addr</i>	Address of the data which needs to be transferred.
<i>len</i>	The data length which needs to be received.

Returns

Returns the length received.

21.2.2.25 **uint32_t SAI_DRV_SendDataBlocking (uint32_t *instance*, uint8_t * *addr*, uint32_t *len*)**

This function sends the data to the Tx FIFO, does not exit until the data is sent to FIFO, and uses the polling way to send audio data.

Parameters

<i>instance</i>	SAI module instance.
<i>addr</i>	Address of the data which needs to be transferred.
<i>len</i>	The data length which need to be sent.

Returns

Returns the length which was sent.

21.2.2.26 `uint32_t SAI_DRV_ReceiveDataBlocking (uint32_t instance, uint8_t * addr, uint32_t len)`

This function receives data from the Rx FIFO, does not exit until the data is received from FIFO, and uses the polling way to send audio data.

Parameters

<i>instance</i>	SAI module instance.
<i>addr</i>	Address of the data which needs to be transferred.
<i>len</i>	The data length which need to be sent.

Returns

Returns the length which was sent.

21.2.2.27 void SAI_DRV_TxRegisterCallback (uint32_t *instance*, sai_callback_t *callback*, void * *callback_param*)

This function tells SAI which function needs to be called after a period length sending. This callback function is used for non-blocking sending.

Parameters

<i>instance</i>	SAI module instance.
<i>callback</i>	Callback function defined by users.
<i>callback_param</i>	The parameter of the callback function.

21.2.2.28 void SAI_DRV_RxRegisterCallback (uint32_t *instance*, sai_callback_t *callback*, void * *callback_param*)

This function tells SAI which function needs to be called after a period length receive. This callback function is used for non-blocking receiving.

Parameters

<i>instance</i>	SAI module instance.
<i>callback</i>	Callback function defined by users.
<i>callback_param</i>	The parameter of the callback function.

21.2.2.29 void SAI_DRV_TxIRQHandler (uint32_t *instance*)

This function sends data in the interrupt and checks the FIFO error.

SAI Peripheral driver

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

21.2.2.30 void SAI_DRV_RxIRQHandler (uint32_t *instance*)

This function receives data in the interrupt and checks the FIFO error.

Parameters

<i>instance</i>	SAI module instance.
-----------------	----------------------

Chapter 22

Secured Digital Host Controller (SDHC)

The Kinetis SDK provides both HAL and Peripheral drivers for the Secure Digital Host Controller (SDHC) block of Kinetis devices.

Modules

- [SDHC Card Definition](#)
This part describes the programming interface of the card data definition for FSL SD host controller.
- [SDHC Card Driver](#)
This part describes the programming interface of the card driver for FSL SD host controller.
- [SDHC Card Related Standard Definition](#)
This part describes the card standard definition.
- [SDHC Data Types](#)
This part describes the SDHC Data Types.
- [SDHC HAL](#)
This part describes the programming interface of the SDHC HAL driver.
- [SDHC Peripheral Driver](#)
This part describes the programming interface of the SDCH Peripheral driver.
- [SDHC Standard Definition](#)
This part describes the host controller standard definition.

22.1 SDHC HAL

This chapter describes the programming interface of the SDHC HAL driver.

SDHC HAL FUNCTION

- static void [SDHC_HAL_SetDmaAddress](#) (uint32_t baseAddr, uint32_t address)
Configures the DMA address.
- static uint32_t [SDHC_HAL_GetDmaAddress](#) (uint32_t baseAddr)
Gets the DMA address.
- static uint32_t [SDHC_HAL_GetBlockSize](#) (uint32_t baseAddr)
Gets the block size configured.
- static void [SDHC_HAL_SetBlockSize](#) (uint32_t baseAddr, uint32_t blockSize)
Sets the block size.
- static void [SDHC_HAL_SetBlockCount](#) (uint32_t baseAddr, uint32_t blockCount)
Sets the block count.
- static uint32_t [SDHC_HAL_GetBlockCount](#) (uint32_t baseAddr)
Gets the block count configured.
- static void [SDHC_HAL_SetCmdArgument](#) (uint32_t baseAddr, uint32_t arg)
Configures the command argument.
- static void [SDHC_HAL_SendCmd](#) (uint32_t baseAddr, uint32_t index, uint32_t flags)
Sends a command.
- static void [SDHC_HAL_SetData](#) (uint32_t baseAddr, uint32_t data)
Fills the data port.
- static uint32_t [SDHC_HAL_GetData](#) (uint32_t baseAddr)
Retrieves the data from the data port.
- static uint32_t [SDHC_HAL_IsCmdInhibit](#) (uint32_t baseAddr)
Checks whether the command inhibit bit is set or not.
- static uint32_t [SDHC_HAL_IsDataInhibit](#) (uint32_t baseAddr)
Checks whether data inhibit bit is set or not.
- static uint32_t [SDHC_HAL_IsDataLineActive](#) (uint32_t baseAddr)
Checks whether data line is active.
- static uint32_t [SDHC_HAL_IsSdClockStable](#) (uint32_t baseAddr)
Checks whether the SD clock is stable or not.
- static uint32_t [SDHC_HAL_IsIpgClockOff](#) (uint32_t baseAddr)
Checks whether the IPG clock is off or not.
- static uint32_t [SDHC_HAL_IsSysClockOff](#) (uint32_t baseAddr)
Checks whether the system clock is off or not.
- static uint32_t [SDHC_HAL_IsPeripheralClockOff](#) (uint32_t baseAddr)
Checks whether the peripheral clock is off or not.
- static uint32_t [SDHC_HAL_IsSdClkOff](#) (uint32_t baseAddr)
Checks whether the SD clock is off or not.
- static uint32_t [SDHC_HAL_IsWriteTransferActive](#) (uint32_t baseAddr)
Checks whether the write transfer is active or not.
- static uint32_t [SDHC_HAL_IsReadTransferActive](#) (uint32_t baseAddr)
Checks whether the read transfer is active or not.
- static uint32_t [SDHC_HAL_IsBuffWriteEnabled](#) (uint32_t baseAddr)
Check whether the buffer write is enabled or not.
- static uint32_t [SDHC_HAL_IsBuffReadEnabled](#) (uint32_t baseAddr)
Checks whether the buffer read is enabled or not.
- static uint32_t [SDHC_HAL_IsCardInserted](#) (uint32_t baseAddr)

- static uint32_t **SDHC_HAL_IsCmdLineLevelHigh** (uint32_t baseAddr)

Checks whether the command line signal is high or not.
- static uint32_t **SDHC_HAL_GetDataLineLevel** (uint32_t baseAddr)

Gets the data line signal level or not.
- static void **SDHC_HAL_SetLedState** (uint32_t baseAddr, sdhc_hal_led_t state)

Sets the LED state.
- static void **SDHC_HAL_SetDataTransferWidth** (uint32_t baseAddr, sdhc_hal_dtw_t dtw)

Sets the data transfer width.
- static bool **SDHC_HAL_IsD3cdEnabled** (uint32_t baseAddr)

Checks whether the DAT3 is taken as card detect pin.
- static void **SDHC_HAL_SetD3cd** (uint32_t baseAddr, bool enable)

Enables the DAT3 as a card detect pin.
- static void **SDHC_HAL_SetEndian** (uint32_t baseAddr, sdhc_hal_endian_t endianMode)

Configures the endian mode.
- static uint32_t **SDHC_HAL_GetCdTestLevel** (uint32_t baseAddr)

Gets the card detect test level.
- static void **SDHC_HAL_SetCdTest** (uint32_t baseAddr, bool enable)

Enables the card detect test.
- static void **SDHC_HAL_SetDmaMode** (uint32_t baseAddr, sdhc_hal_dma_mode_t dmaMode)

Sets the DMA mode.
- static void **SDHC_HAL_SetStopAtBlockGap** (uint32_t baseAddr, bool enable)

Enables stop at the block gap.
- static void **SDHC_HAL_SetContinueRequest** (uint32_t baseAddr)

Restarts a transaction which has stopped at the block gap.
- static void **SDHC_HAL_SetReadWaitCtrl** (uint32_t baseAddr, bool enable)

Enables the read wait control for the SDIO cards.
- static void **SDHC_HAL_SetIntStopAtBlockGap** (uint32_t baseAddr, bool enable)

Enables stop at the block gap requests.
- static void **SDHC_HAL_SetWakeupOnCardInt** (uint32_t baseAddr, bool enable)

Enables wakeup event on the card interrupt.
- static void **SDHC_HAL_SetWakeupOnCardInsertion** (uint32_t baseAddr, bool enable)

Enables wakeup event on the card insertion.
- static void **SDHC_HAL_SetWakeupOnCardRemoval** (uint32_t baseAddr, bool enable)

Enables wakeup event on card removal.
- static void **SDHC_HAL_SetIpClock** (uint32_t baseAddr, bool enable)

Enables the IPG clock and no automatic clock gating off.
- static void **SDHC_HAL_SetSysClock** (uint32_t baseAddr, bool enable)

Enables the system clock and no automatic clock gating off.
- static void **SDHC_HAL_SetPeripheralClock** (uint32_t baseAddr, bool enable)

Enables the peripheral clock and no automatic clock gating off.
- static void **SDHC_HAL_SetSdClock** (uint32_t baseAddr, bool enable)

Enables the SD clock.
- static void **SDHC_HAL_SetClockDivisor** (uint32_t baseAddr, uint32_t divisor)

Sets the SD clock frequency divisor.
- static void **SDHC_HAL_SetClockFrequency** (uint32_t baseAddr, uint32_t frequency)

Sets the SD clock frequency select.
- static void **SDHC_HAL_SetDataTimeout** (uint32_t baseAddr, uint32_t timeout)

Sets the data timeout counter value.
- static uint32_t **SDHC_HAL_GetIntFlags** (uint32_t baseAddr)

Gets the current interrupt status.

SDHC HAL

- static void **SDHC_HAL_ClearIntFlags** (uint32_t baseAddr, uint32_t mask)
Clears a specified interrupt status.
- static uint32_t **SDHC_HAL_GetIntSignal** (uint32_t baseAddr)
Gets the currently enabled interrupt signal.
- static uint32_t **SDHC_HAL_GetIntState** (uint32_t baseAddr)
Gets the currently enabled interrupt state.
- static uint32_t **SDHC_HAL_GetAc12Error** (uint32_t baseAddr)
Gets the auto cmd12 error.
- static uint32_t **SDHC_HAL_GetMaxBlockLength** (uint32_t baseAddr)
Gets the maximum block length supported.
- static uint32_t **SDHC_HAL_DoesHostSupportAdma** (uint32_t baseAddr)
Checks whether the ADMA is supported.
- static uint32_t **SDHC_HAL_DoesHostSupportHighspeed** (uint32_t baseAddr)
Checks whether the high speed is supported.
- static uint32_t **SDHC_HAL_DoesHostSupportDma** (uint32_t baseAddr)
Checks whether the DMA is supported.
- static uint32_t **SDHC_HAL_DoesHostSupportSuspendResume** (uint32_t baseAddr)
Checks whether the suspend/resume is supported.
- static uint32_t **SDHC_HAL_DoesHostSupportV330** (uint32_t baseAddr)
Checks whether the voltage 3.3 is supported.
- static uint32_t **SDHC_HAL_DoesHostSupportV300** (uint32_t baseAddr)
Checks whether the voltage 3.0 is supported.
- static uint32_t **SDHC_HAL_DoesHostSupportV180** (uint32_t baseAddr)
Checks whether the voltage 1.8 is supported.
- static void **SDHC_HAL_SetWriteWatermarkLevel** (uint32_t baseAddr, uint32_t watermark)
Sets the watermark for writing.
- static void **SDHC_HAL_SetReadWatermarkLevel** (uint32_t baseAddr, uint32_t watermark)
Sets the watermark for reading.
- static void **SDHC_HAL_SetForceEventFlags** (uint32_t baseAddr, uint32_t mask)
Sets the force events according to the given mask.
- static uint32_t **SDHC_HAL_IsAdmaLengthMismatchError** (uint32_t baseAddr)
Checks whether the ADMA error is length mismatch.
- static bool **SDHC_HAL_IsSdClockOff** (uint32_t baseAddr)
Checks the SD clock.
- static uint32_t **SDHC_HAL_GetAdmaErrorState** (uint32_t baseAddr)
Returns the state of the ADMA error.
- static uint32_t **SDHC_HAL_IsAdmaDescriptorError** (uint32_t baseAddr)
Checks whether the ADMA error is a descriptor error.
- static void **SDHC_HAL_SetAdmaAddress** (uint32_t baseAddr, uint32_t address)
Sets the ADMA address.
- static void **SDHC_HAL_SetExternalDmaRequest** (uint32_t baseAddr, bool enable)
Enables the external DMA request.
- static void **SDHC_HAL_SetExactBlockNumber** (uint32_t baseAddr, bool enable)
Enables the exact block number for the SDIO CMD53.
- static void **SDHC_HAL_SetBootAckTimeout** (uint32_t baseAddr, uint32_t timeout)
Sets the timeout value for the boot ACK.
- static void **SDHC_HAL_SetBootAck** (uint32_t baseAddr, bool enable)
Enables the boot ACK.
- static void **SDHC_HAL_SetBootMode** (uint32_t baseAddr, sdhc_hal_mmcboot_t mode)
Configures the boot mode.
- static void **SDHC_HAL_SetFastboot** (uint32_t baseAddr, bool enable)

- static void **SDHC_HAL_SetAutoStopAtBlockGap** (uint32_t baseAddr, bool enable)

Enables the fast boot.
- static void **SDHC_HAL_SetBootBlockCount** (uint32_t baseAddr, uint32_t blockCount)

Enables the automatic stop at the block gap.
- static uint32_t **SDHC_HAL_GetSpecificationVersion** (uint32_t baseAddr)

Configures the the block count for the boot.
- static uint32_t **SDHC_HAL_GetVendorVersion** (uint32_t baseAddr)

Gets a specification version.
- uint32_t **SDHC_HAL.GetResponse** (uint32_t baseAddr, uint32_t index)

Gets the command response.
- void **SDHC_HAL_SetIntSignal** (uint32_t baseAddr, bool enable, uint32_t mask)

Enables the specified interrupts.
- void **SDHC_HAL_SetIntState** (uint32_t baseAddr, bool enable, uint32_t mask)

Enables the specified interrupt state.
- uint32_t **SDHC_HAL_Reset** (uint32_t baseAddr, uint32_t type, uint32_t timeout)

Performs an SDHC reset.
- uint32_t **SDHC_HAL_InitCard** (uint32_t baseAddr, uint32_t timeout)

Sends 80 clocks to the card to initialize the card.
- IRQn_Type **SDHC_HAL_GetIrqId** (uint32_t baseAddr)

Gets the IRQ ID for a given host controller.
- void **SDHC_HAL_Init** (uint32_t baseAddr)

Initializes the SDHC HAL.

22.1.1 Function Documentation

22.1.1.1 static void **SDHC_HAL_SetDmaAddress** (*uint32_t baseAddr, uint32_t address*) [**inline**], [**static**]

Parameters

<i>baseAddr</i>	SDHC base address
<i>address</i>	the DMA address

22.1.1.2 static uint32_t **SDHC_HAL_GetDmaAddress** (*uint32_t baseAddr*) [**inline**], [**static**]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

the DMA address

SDHC HAL

22.1.1.3 **static uint32_t SDHC_HAL_GetBlockSize(uint32_t *baseAddr*) [inline], [static]**

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

the block size already configured

22.1.1.4 static void SDHC_HAL_SetBlockSize (*uint32_t baseAddr, uint32_t blockSize*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>blockSize</i>	the block size

22.1.1.5 static void SDHC_HAL_SetBlockCount (*uint32_t baseAddr, uint32_t blockCount*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>blockCount</i>	the block count

22.1.1.6 static uint32_t SDHC_HAL_GetBlockCount (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

the block count already configured

22.1.1.7 static void SDHC_HAL_SetCmdArgument (*uint32_t baseAddr, uint32_t arg*) [inline], [static]

SDHC HAL

Parameters

<i>baseAddr</i>	SDHC base address
<i>arg</i>	the command argument

22.1.1.8 static void SDHC_HAL_SendCmd (uint32_t *baseAddr*, uint32_t *index*, uint32_t *flags*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>index</i>	command index
<i>flags</i>	transfer type flags

22.1.1.9 static void SDHC_HAL_SetData (uint32_t *baseAddr*, uint32_t *data*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>data</i>	the data about to be sent

22.1.1.10 static uint32_t SDHC_HAL_GetData (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

data the data read

22.1.1.11 static uint32_t SDHC_HAL_IsCmdInhibit (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if command inhibit, 0 if not.

22.1.1.12 static uint32_t SDHC_HAL_IsDataInhibit (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if data inhibit, 0 if not.

22.1.1.13 static uint32_t SDHC_HAL_IsDataLineActive (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's active, 0 if not.

22.1.1.14 static uint32_t SDHC_HAL_IsSdClockStable (uint32_t *baseAddr*) [inline], [static]

Parameters

SDHC HAL

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's stable, 0 if not.

22.1.1.15 static uint32_t SDHC_HAL_IsLpClockOff(uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's off, 0 if not.

22.1.1.16 static uint32_t SDHC_HAL_IsSysClockOff(uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's off, 0 if not.

22.1.1.17 static uint32_t SDHC_HAL_IsPeripheralClockOff(uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address.
-----------------	--------------------

Returns

1 if it's off, 0 if not.

22.1.1.18 static uint32_t SDHC_HAL_IsSdClkOff(uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's off, 0 if not.

22.1.1.19 static uint32_t SDHC_HAL_IsWriteTransferActive (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's active, 0 if not.

22.1.1.20 static uint32_t SDHC_HAL_IsReadTransferActive (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's off, 0 if not.

22.1.1.21 static uint32_t SDHC_HAL_IsBuffWriteEnabled (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's isEnableddd, 0 if not.

SDHC HAL

22.1.1.22 **static uint32_t SDHC_HAL_IsBuffReadEnabled (*uint32_t baseAddr*)**
[**inline**], [**static**]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's isEnableddd, 0 if not.

22.1.1.23 static uint32_t SDHC_HAL_IsCardInserted (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address.
-----------------	--------------------

Returns

1 if it's inserted, 0 if not.

22.1.1.24 static uint32_t SDHC_HAL_IsCmdLineLevelHigh (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

1 if it's high, 0 if not.

22.1.1.25 static uint32_t SDHC_HAL_GetDataLineLevel (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

[7:0] data line signal level

SDHC HAL

22.1.1.26 static void SDHC_HAL_SetLedState(uint32_t *baseAddr*, sdhc_hal_led_t *state*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>state</i>	the LED state

22.1.1.27 static void SDHC_HAL_SetDataTransferWidth (uint32_t *baseAddr*, sdhc_hal_dtw_t *dtw*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>dtw</i>	data transfer width

22.1.1.28 static bool SDHC_HAL_IsD3cdEnabled (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if DAT3 as card detect pin is enabled

22.1.1.29 static void SDHC_HAL_SetD3cd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable DAT3 as card detect pin

22.1.1.30 static void SDHC_HAL_SetEndian (uint32_t *baseAddr*, sdhc_hal_endian_t *endianMode*) [inline], [static]

SDHC HAL

Parameters

<i>baseAddr</i>	SDHC base address
<i>endianMode</i>	endian mode

22.1.1.31 static uint32_t SDHC_HAL_GetCdTestLevel (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

card detect test level

22.1.1.32 static void SDHC_HAL_SetCdTest (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable card detect signal for test purpose

22.1.1.33 static void SDHC_HAL_SetDmaMode (uint32_t *baseAddr*, sdhc_hal_dma_mode_t *dmaMode*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>dmaMode</i>	the DMA mode

22.1.1.34 static void SDHC_HAL_SetStopAtBlockGap (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to stop at block gap request

22.1.1.35 static void SDHC_HAL_SetContinueRequest (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

22.1.1.36 static void SDHC_HAL_SetReadWaitCtrl (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable read wait control

22.1.1.37 static void SDHC_HAL_SetIntStopAtBlockGap (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable interrupt at block gap

22.1.1.38 static void SDHC_HAL_SetWakeupOnCardInt (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable wakeup event on card interrupt

SDHC HAL

22.1.1.39 **static void SDHC_HAL_SetWakeupOnCardInsertion (uint32_t *baseAddr*, bool *enable*) [inline], [static]**

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable wakeup event on card insertion

22.1.1.40 static void SDHC_HAL_SetWakeupOnCardRemoval (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable wakeup event on card removal

22.1.1.41 static void SDHC_HAL_SetIpgClock (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable IPG clock

22.1.1.42 static void SDHC_HAL_SetSysClock (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable SYS clock

22.1.1.43 static void SDHC_HAL_SetPeripheralClock (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable Peripheral clock

**22.1.1.44 static void SDHC_HAL_SetSdClock (uint32_t *baseAddr*, bool *enable*)
[inline], [static]**

It should be disabled before changing the SD clock frequency.

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable SD clock or not

22.1.1.45 static void SDHC_HAL_SetClockDivisor (uint32_t *baseAddr*, uint32_t *divisor*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>divisor</i>	the divisor

22.1.1.46 static void SDHC_HAL_SetClockFrequency (uint32_t *baseAddr*, uint32_t *frequency*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>frequency</i>	the frequency selector

22.1.1.47 static void SDHC_HAL_SetDataTimeout (uint32_t *baseAddr*, uint32_t *timeout*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>timeout</i>	Data timeout counter value

22.1.1.48 static uint32_t SDHC_HAL_GetIntFlags (uint32_t *baseAddr*) [inline], [static]

Parameters

SDHC HAL

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

current interrupt flags

22.1.1.49 static void SDHC_HAL_ClearIntFlags (uint32_t *baseAddr*, uint32_t *mask*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>mask</i>	to specify interrupts' flags to be cleared

22.1.1.50 static uint32_t SDHC_HAL_GetIntSignal (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

currently enabled interrupt signal

22.1.1.51 static uint32_t SDHC_HAL_GetIntState (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

currently enabled interrupts' state

22.1.1.52 static uint32_t SDHC_HAL_GetAc12Error (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

auto cmd12 error status

22.1.1.53 static uint32_t SDHC_HAL_GetMaxBlockLength (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

the maximum block length support

22.1.1.54 static uint32_t SDHC_HAL_DoesHostSupportAdma (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if ADMA is supported

22.1.1.55 static uint32_t SDHC_HAL_DoesHostSupportHighspeed (uint32_t *baseAddr*) [inline], [static]

Parameters

SDHC HAL

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if high speed is supported

**22.1.1.56 static uint32_t SDHC_HAL_DoesHostSupportDma (uint32_t *baseAddr*)
[inline], [static]**

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if high speed is supported

**22.1.1.57 static uint32_t SDHC_HAL_DoesHostSupportSuspendResume (uint32_t
baseAddr) [inline], [static]**

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if suspend and resume is supported

**22.1.1.58 static uint32_t SDHC_HAL_DoesHostSupportV330 (uint32_t *baseAddr*)
[inline], [static]**

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if voltage 3.3 is supported

**22.1.1.59 static uint32_t SDHC_HAL_DoesHostSupportV300 (uint32_t *baseAddr*)
[inline], [static]**

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if voltage 3.0 is supported

22.1.1.60 static uint32_t SDHC_HAL_DoesHostSupportV180 (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if voltage 1.8 is supported

22.1.1.61 static void SDHC_HAL_SetWriteWatermarkLevel (uint32_t *baseAddr*, uint32_t *watermark*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>watermark</i>	for writing

22.1.1.62 static void SDHC_HAL_SetReadWatermarkLevel (uint32_t *baseAddr*, uint32_t *watermark*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>watermark</i>	for reading

22.1.1.63 static void SDHC_HAL_SetForceEventFlags (uint32_t *baseAddr*, uint32_t *mask*) [inline], [static]

SDHC HAL

Parameters

<i>baseAddr</i>	SDHC base address
<i>mask</i>	to specify the force events' flags to be set

22.1.1.64 static uint32_t SDHC_HAL_IsAdmaLengthMismatchError (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if ADMA error is length mismatch

22.1.1.65 static bool SDHC_HAL_IsSdClockOff (uint32_t *baseAddr*) [inline], [static]

Checks whether the clock to the SD is enabled.

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

true if enabled

22.1.1.66 static uint32_t SDHC_HAL_GetAdmaErrorState (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

error state

22.1.1.67 static uint32_t SDHC_HAL_IsAdmaDescriptionError (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

if ADMA error is descriptor error

22.1.1.68 static void SDHC_HAL_SetAdmaAddress (uint32_t *baseAddr*, uint32_t *address*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>address</i>	for ADMA transfer

22.1.1.69 static void SDHC_HAL_SetExternalDmaRequest (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to external DMA

22.1.1.70 static void SDHC_HAL_SetExactBlockNumber (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable exact block number block read for SDIO CMD53

22.1.1.71 static void SDHC_HAL_SetBootAckTimeout (uint32_t *baseAddr*, uint32_t *timeout*) [inline], [static]

SDHC HAL

Parameters

<i>baseAddr</i>	SDHC base address
<i>timeout</i>	boot ack time out counter value

22.1.1.72 static void SDHC_HAL_SetBootAck (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable boot ack mode

22.1.1.73 static void SDHC_HAL_SetBootMode (uint32_t *baseAddr*, sdhc_hal_mmcboot_t *mode*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>mode</i>	the boot mode

22.1.1.74 static void SDHC_HAL_SetFastboot (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable fast boot

22.1.1.75 static void SDHC_HAL_SetAutoStopAtBlockGap (uint32_t *baseAddr*, bool *enable*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	to enable auto stop at block gap function, when boot.

22.1.1.76 static void SDHC_HAL_SetBootBlockCount (*uint32_t baseAddr, uint32_t blockCount*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
<i>blockCount</i>	the block count for boot

22.1.1.77 static uint32_t SDHC_HAL_GetSpecificationVersion (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

specification version

22.1.1.78 static uint32_t SDHC_HAL_GetVendorVersion (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

vendor version

22.1.1.79 uint32_t SDHC_HAL.GetResponse (*uint32_t baseAddr, uint32_t index*)

SDHC HAL

Parameters

<i>baseAddr</i>	SDHC base address
<i>index</i>	of response register, range from 0 to 3

22.1.1.80 void SDHC_HAL_SetIntSignal (uint32_t *baseAddr*, bool *enable*, uint32_t *mask*)

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	enable or disable
<i>mask</i>	to specify interrupts to be isEnableddd

22.1.1.81 void SDHC_HAL_SetIntState (uint32_t *baseAddr*, bool *enable*, uint32_t *mask*)

Parameters

<i>baseAddr</i>	SDHC base address
<i>enable</i>	enable or disable
<i>mask</i>	to specify interrupts' state to be enabled

22.1.1.82 uint32_t SDHC_HAL_Reset (uint32_t *baseAddr*, uint32_t *type*, uint32_t *timeout*)

Parameters

<i>baseAddr</i>	SDHC base address
<i>type</i>	the type of reset
<i>timeout</i>	timeout for reset

Returns

0 on success, else on error

22.1.1.83 uint32_t SDHC_HAL_InitCard (uint32_t *baseAddr*, uint32_t *timeout*)

Parameters

<i>baseAddr</i>	SDHC base address
<i>timeout</i>	timeout for initialize card

Returns

0 on success, else on error

22.1.1.84 IRQn_Type SDHC_HAL_GetIrqId (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

Returns

IRQ number for specific SDHC instance

22.1.1.85 void SDHC_HAL_Init (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	SDHC base address
-----------------	-------------------

22.2 SDHC Peripheral Driver

This chapter describes the programming interface of the SDCH Peripheral driver.

SDHC PD FUNCTION

- `sdhc_status_t SDHC_DRV_Init (uint32_t instance, sdhc_host_t *host, const sdhc_user_config_t *config)`
Initializes the Host controller by a specific instance index.
- `void SDHC_DRV_Shutdown (uint32_t instance)`
Destroys the host controller.
- `sdhc_status_t SDHC_DRV_DetectCard (uint32_t instance)`
Checks whether the card is present on specified host controller.
- `sdhc_status_t SDHC_DRV_ConfigClock (uint32_t instance, uint32_t clock)`
Sets the clock frequency of the host controller.
- `sdhc_status_t SDHC_DRV_SetBusWidth (uint32_t instance, sdhc_buswidth_t busWidth)`
Sets the bus width of the host controller.
- `sdhc_status_t SDHC_DRV_IssueRequestBlocking (uint32_t instance, sdhc_request_t *req, uint32_t timeoutInMs)`
Issues the request on a specific host controller and returns on completion.
- `void SDHC_DRV_DoIrq (uint32_t instance)`
IRQ handler for SDHC.

22.2.0.86 SDHC Peripheral Driver

Overview

The SDHC driver configures the secure digital host controller and provides an easy way to operate the SDHC module.

Initialization

To initialize the SDHC module, call the `SDHC_DRV_Init()` function and pass in the configuration data structure.

This is an example code to initialize and configure the driver:

```
// Define device configuration.  
sdhc_user_config_t config = {0};  
  
config.clock = 0;  
  
// Initialize  
if (SDHC_DRV_Init(instance, &host, &config) != kStatus_SDHC_NoError)  
{  
    /* error occurs */  
}  
else  
{
```

```
    /* SDHC has been successfully initialized */
}
```

Issuing a request to the card

The SDHC driver provides a simple way to send commands to and retrieve response/data from the card. This is a example to send the SD_SWITCH command to the card.

```
sdhc_request_t req = {0};
sdhc_data_t data = {0};

req.cmdIndex = kSdSwitch;
req.argument = mode << 31 | 0x00FFFFFF;
req.argument &= ~((0xF) << (group * 4));
req.argument |= value << (group * 4);
req.flags = FSL_SDHC_REQ_FLAGS_DATA_READ;
req.respType = kSdhcRespTypeR1;

data.blockSize = 64;
data.blockCount = 1;
data.buffer = response;

/* link data, command with request */
data.req = &req;
req.data = &data;
if (kStatus_SDHC_NoError != SDHC_DRV_IssueRequestBlocking(
    card->hostInstance,
    &req,
    FSL_SDCARD_REQUEST_TIMEOUT);) /* issue request */

{
    /* error occurs */
}
else
{
    /* request has been issued successfully */
}
```

22.2.1 Function Documentation

22.2.1.1 **sdhc_status_t SDHC_DRV_Init (uint32_t *instance*, sdhc_host_t * *host*, const sdhc_user_config_t * *config*)**

This function initializes the SDHC module according to the given initialization configuration structure including the clock frequency, bus width, and card detect callback.

Parameters

SDHC Peripheral Driver

<i>instance</i>	the specific instance index
<i>host</i>	pointer to a place storing the <code>sdhc_host_t</code> structure
<i>config</i>	initialization configuration data

Returns

`kStatus_SDHC_NoError` if success

22.2.1.2 `void SDHC_DRV_Shutdown (uint32_t instance)`

Parameters

<i>instance</i>	the instance index of host controller
-----------------	---------------------------------------

22.2.1.3 `sdhc_status_t SDHC_DRV_DetectCard (uint32_t instance)`

This function checks if there's a card inserted to the SDHC.

Parameters

<i>instance</i>	the instance index of host controller
-----------------	---------------------------------------

Returns

`kStatus_SDHC_NoError` on success

22.2.1.4 `sdhc_status_t SDHC_DRV_ConfigClock (uint32_t instance, uint32_t clock)`

Parameters

<i>instance</i>	the instance index of host controller
<i>clock</i>	the desired frequency to be set to controller

Returns

`kStatus_SDHC_NoError` on success

22.2.1.5 `sdhc_status_t SDHC_DRV_SetBusWidth (uint32_t instance, sdhc_buswidth_t busWidth)`

Parameters

<i>instance</i>	the instance index of host controller
<i>busWidth</i>	the desired bus width to be set to controller

Returns

kStatus_SDHC_NoError on success

22.2.1.6 **sdhc_status_t SDHC_DRV_IssueRequestBlocking (uint32_t *instance*, sdhc_request_t * *req*, uint32_t *timeoutInMs*)**

This function issues the request to the card on a specific SDHC. The command is sent and is blocked as long as the response/data is sending back from the card.

Parameters

<i>instance</i>	the instance index of host controller
<i>req</i>	the pointer to the request
<i>timeoutInMs</i>	timeout value in microseconds

Returns

kStatus_SDHC_NoError on success

22.2.1.7 **void SDHC_DRV_Dolrq (uint32_t *instance*)**

This function deals with IRQs on the given host controller.

Parameters

<i>instance</i>	the instance index of host controller
-----------------	---------------------------------------

SDHC Data Types

22.3 SDHC Data Types

This chapter describes the SDHC Data Types.

Data Structures

- struct [sdhc_user_config_t](#)
SDHC Initialization Configuration Structure. [More...](#)
- struct [sdhc_host_t](#)
SDHC Host Device Structure. [More...](#)
- struct [sdhc_data_t](#)
SDHC Data Structure. [More...](#)
- struct [sdhc_request_t](#)
SDHC Request Structure. [More...](#)

Enumerations

- enum `sdhc_status_t` {

kStatus_SDHC_NoError = 0,

kStatus_SDHC_InitFailed,

kStatus_SDHC_SetClockFailed,

kStatus_SDHC_SetCardToIdle,

kStatus_SDHC_SetCardBlockSizeFailed,

kStatus_SDHC_SendAppOpCondFailed,

kStatus_SDHC_AllSendCidFailed,

kStatus_SDHC_SendRcaFailed,

kStatus_SDHC_SendCsdFailed,

kStatus_SDHC_SendScrFailed,

kStatus_SDHC_SelectCardFailed,

kStatus_SDHC_SwitchHighSpeedFailed,

kStatus_SDHC_SetCardWideBusFailed,

kStatus_SDHC_SetBusWidthFailed,

kStatus_SDHC_SendCardStatusFailed,

kStatus_SDHC_StopTransmissionFailed,

kStatus_SDHC_CardEraseBlocksFailed,

kStatus_SDHC_InvalidIORange,

kStatus_SDHC_BlockSizeNotSupportError,

kStatus_SDHC_HostIsAlreadyInitiated,

kStatus_SDHC_HostNotSupport,

kStatus_SDHC_HostIsBusyError,

kStatus_SDHC_DataPrepareError,

kStatus_SDHC_WaitTimeoutError,

kStatus_SDHC_OutOfMemory,

kStatus_SDHC_IoError,

kStatus_SDHC_CmdIoError,

kStatus_SDHC_DataIoError,

kStatus_SDHC_InvalidParameter,

kStatus_SDHC_RequestFailed,

kStatus_SDHC_RequestCardStatusError,

kStatus_SDHC_SwitchFailed,

kStatus_SDHC_NotSupportYet,

kStatus_SDHC_TimeoutError,

kStatus_SDHC_CardNotSupport,

kStatus_SDHC_CmdError,

kStatus_SDHC_DataError,

kStatus_SDHC_DmaAddressError,

kStatus_SDHC_Failed,

kStatus_SDHC_NoMedium,

kStatus_SDHC_UnknownStatus
 }

• enum `sdhc_cd_type_t` {

SDHC Data Types

```
kSdhcCardDetectGpio = 1,  
kSdhcCardDetectDat3,  
kSdhcCardDetectCdPin,  
kSdhcCardDetectPollDat3,  
kSdhcCardDetectPollCd }  
• enum sdhc_power_mode_t {  
    kSdhcPowerModeRunning = 0,  
    kSdhcPowerModeSuspended,  
    kSdhcPowerModeStopped }  
• enum sdhc_buswidth_t {  
    kSdhcBusWidth1Bit = 1,  
    kSdhcBusWidth4Bit,  
    kSdhcBusWidth8Bit }  
• enum sdhc_transfer_mode_t {  
    kSdhcTransModePio = 1,  
    kSdhcTransModeSdma,  
    kSdhcTransModeAdma1,  
    kSdhcTransModeAdma2 }  
• enum sdhc_resp_type_t {  
    kSdhcRespTypeNone = 0,  
    kSdhcRespTypeR1,  
    kSdhcRespTypeR1b,  
    kSdhcRespTypeR2,  
    kSdhcRespTypeR3,  
    kSdhcRespTypeR4,  
    kSdhcRespTypeR5,  
    kSdhcRespTypeR5b,  
    kSdhcRespTypeR6,  
    kSdhcRespTypeR7 }
```

22.3.1 Data Structure Documentation

22.3.1.1 struct sdhc_user_config_t

Defines the configuration data structure to initialize the SDHC.

Data Fields

- uint32_t **clock**
Clock rate.
- sdhc_transfer_mode_t **transMode**
SDHC transfer mode.
- sdhc_cd_type_t **cdType**
Card detection type.
- void(* **cardDetectCallback**)(bool inserted)

- `void(* cardIntCallback)(void)`
Callback function for card detect occurs.
- `void(* blockGapCallback)(void)`
Callback function for block gap occurs.

22.3.1.2 struct sdhc_host_t

Defines the Host device structure which includes both the static and the runtime SDHC information.

Data Fields

- `uint32_t instance`
Host instance index.
- `sdhc_cd_type_t cdType`
Host controller card detection type.
- `sdhc_hal_endian_t endian`
Endian mode the host's working at.
- `uint32_t swFeature`
Host controller driver features.
- `uint32_t flags`
Host flags.
- `uint32_t busWidth`
Current busWidth.
- `uint32_t caps`
Host capability.
- `uint32_t ocrSupported`
Supported OCR.
- `uint32_t clock`
Current clock frequency.
- `sdhc_power_mode_t powerMode`
Current power mode.
- `uint32_t maxClock`
Maximum clock supported.
- `uint32_t maxBlockSize`
Maximum block size supported.
- `uint32_t maxBlockCount`
Maximum block count supported.
- `uint32_t * admaTableAddress`
ADMA table address.
- `uint32_t admaTableMaxEntries`
Maximum entries can be held in table.
- `struct SdhcRequest * currentReq`
Associated request.
- `void(* cardIntCallback)(void)`
Callback function for card interrupt occurs.
- `void(* cardDetectCallback)(bool inserted)`
Callback function for card detect occurs.
- `void(* blockGapCallback)(void)`

SDHC Data Types

Callback function for block gap occurs.

22.3.1.3 struct sdhc_data_t

Defines the SDHC data structure including the block size/count and flags.

Data Fields

- struct SdhcRequest * **req**
Associated request.
- uint32_t **blockSize**
Block size.
- uint32_t **blockCount**
Block count.
- uint32_t **bytesTransferred**
Transferred buffer.
- uint32_t * **buffer**
Data buffer.

22.3.1.4 struct sdhc_request_t

Defines the SDHC request structure including the command index, argument, flags, response, and data.

Data Fields

- uint32_t **cmdIndex**
Command index.
- uint32_t **argument**
Command argument.
- uint32_t **flags**
Flags.
- **sdhc_resp_type_t respType**
Response type.
- volatile uint32_t **error**
Command error code.
- uint32_t **cardErrStatus**
Card error status from response 1.
- uint32_t **response [4]**
Response for this command.
- semaphore_t * **complete**
Request completion sync object.
- struct SdhcData * **data**
Data associated with request.

22.3.2 Enumeration Type Documentation

22.3.2.1 enum sdhc_status_t

Enumerator

kStatus_SDHC_NoError No error.
kStatus_SDHC_InitFailed Driver initialization failed.
kStatus_SDHC_SetClockFailed Failed to set clock of host controller.
kStatus_SDHC_SetCardToIdle Failed to set card to idle.
kStatus_SDHC_SetCardBlockSizeFailed Failed to set card block size.
kStatus_SDHC_SendAppOpCondFailed Failed to send app_op_cond command.
kStatus_SDHC_AllSendCidFailed Failed to send all_send_cid command.
kStatus_SDHC_SendRcaFailed Failed to send send_rca command.
kStatus_SDHC_SendCsdFailed Failed to send send_csd command.
kStatus_SDHC_SendScrFailed Failed to send send_scr command.
kStatus_SDHC_SelectCardFailed Failed to send select_card command.
kStatus_SDHC_SwitchHighSpeedFailed Failed to switch to high speed mode.
kStatus_SDHC_SetCardWideBusFailed Failed to set card's bus mode.
kStatus_SDHC_SetBusWidthFailed Failed to set host's bus mode.
kStatus_SDHC_SendCardStatusFailed Failed to send card status.
kStatus_SDHC_StopTransmissionFailed Failed to stop transmission.
kStatus_SDHC_CardEraseBlocksFailed Failed to erase blocks.
kStatus_SDHC_InvalidIORange Invalid read/write/erase address range.
kStatus_SDHC_BlockSizeNotSupportError Unsupported block size.
kStatus_SDHC_HostIsAlreadyInitiated Host controller is already initialized.
kStatus_SDHC_HostNotSupport Host not error.
kStatus_SDHC_HostIsBusyError Bus busy error.
kStatus_SDHC_DataPrepareError Data preparation error.
kStatus_SDHC_WaitTimeoutError Wait timeout error.
kStatus_SDHC_OutOfMemory Out of memory error.
kStatus_SDHC_IoError General IO error.
kStatus_SDHC_CmdIoError CMD I/O error.
kStatus_SDHC_DataIoError Data I/O error.
kStatus_SDHC_InvalidParameter Invalid parameter error.
kStatus_SDHC_RequestFailed Request failed.
kStatus_SDHC_RequestCardStatusError Status error.
kStatus_SDHC_SwitchFailed Switch failed.
kStatus_SDHC_NotSupportYet Not support.
kStatus_SDHC_TimeoutError Timeout error.
kStatus_SDHC_CardNotSupport Card does not support.
kStatus_SDHC_CmdError CMD error.
kStatus_SDHC_DataError Data error.
kStatus_SDHC_DmaAddressError DMA address error.
kStatus_SDHC_Failed General failed.

SDHC Data Types

kStatus_SDHC_NoMedium No medium error.

kStatus_SDHC_UnknownStatus Unknown if card is present.

22.3.2.2 enum sdhc_cd_type_t

Enumerator

kSdhcCardDetectGpio Use GPIO for card detection.

kSdhcCardDetectDat3 Use DAT3 for card detection.

kSdhcCardDetectCdPin Use host controller dedicate CD pin for card detection.

kSdhcCardDetectPollDat3 Poll DAT3 for card detection.

kSdhcCardDetectPollCd Poll host controller dedicate CD pin for card detection.

22.3.2.3 enum sdhc_power_mode_t

Enumerator

kSdhcPowerModeRunning SDHC is running.

kSdhcPowerModeSuspended SDHC is suspended.

kSdhcPowerModeStopped SDHC is stopped.

22.3.2.4 enum sdhc_buswidth_t

Enumerator

kSdhcBusWidth1Bit 1-bit bus width

kSdhcBusWidth4Bit 4-bit bus width

kSdhcBusWidth8Bit 8-bit bus width

22.3.2.5 enum sdhc_transfer_mode_t

Enumerator

kSdhcTransModePio Transfer mode: PIO.

kSdhcTransModeSdma Transfer mode: SDMA.

kSdhcTransModeAdma1 Transfer mode: ADMA1.

kSdhcTransModeAdma2 Transfer mode: ADMA2.

22.3.2.6 enum sdhc_resp_type_t

Enumerator

- kSdhcRespTypeNone* Response type: none.
- kSdhcRespTypeR1* Response type: R1.
- kSdhcRespTypeR1b* Response type: R1b.
- kSdhcRespTypeR2* Response type: R2.
- kSdhcRespTypeR3* Response type: R3.
- kSdhcRespTypeR4* Response type: R4.
- kSdhcRespTypeR5* Response type: R5.
- kSdhcRespTypeR5b* Response type: R5b.
- kSdhcRespTypeR6* Response type: R6.
- kSdhcRespTypeR7* Response type: R7.

SDHC Standard Definition

22.4 SDHC Standard Definition

This chapter describes the host controller standard definition.

Macros

- #define **SDHC_DMA_ADDRESS** (0x00U)
SDHC DMA ADDRESS REG.
- #define **SDHC_BLOCK_SIZE** (0x04U)
SDHC BLOCK SIZE REG.
- #define **SDHC_BLOCK_COUNT** (0x06U)
SDHC BLOCK COUNT REG.
- #define **SDHC_ARGUMENT** (0x08U)
SDHC ARGUMENT REG.
- #define **SDHC_TRANSFER_MODE** (0x0C)
SDHC TRANSFER MODE REG.
- #define **SDHC_TRNSM_DMA_EN** (0x01U)
SDHC TRANSFER MODE DMA ENABLE BIT.
- #define **SDHC_TRNSM_BLKCNT_EN** (0x02U)
SDHC TRANSFER MODE BLOCK COUNT ENABLE BIT.
- #define **SDHC_TRNSM_AUTOCMD12** (0x04U)
SDHC TRANSFER MODE AUTO CMD12 BIT.
- #define **SDHC_TRNSM_AUTOCMD23** (0x08U)
SDHC TRANSFER MODE AUTO CMD23 BIT.
- #define **SDHC_TRNSM_READ** (0x10U)
SDHC TRANSFER MODE READ DATA BIT.
- #define **SDHC_TRNSM_MULTI** (0x20U)
SDHC TRANSFER MODE MULTIBLOCK BIT.
- #define **SDHC_COMMAND** (0x0E)
SDHC COMMAND REG.
- #define **SDHC_CMD_RESPTYPE_LSF** (0U)
SDHC COMMAND RESPONSE TYPE SHIFT.
- #define **SDHC_CMD_RESPTYPE_MASK** (0x03U)
SDHC COMMAND RESPONSE MASK.
- #define **SDHC_CMD_CRC_CHK** (0x08U)
SDHC COMMAND CRC CHECKING BIT.
- #define **SDHC_CMD_INDEX_CHK** (0x10U)
SDHC COMMAND INDEX CHECKING BIT.
- #define **SDHC_CMD_DATA_PRSNT** (0x20U)
SDHC COMMAND DATA PRESENT BIT.
- #define **SDHC_CMD_CMDTYPE_LSF** (6U)
SDHC COMMAND COMMAND TYPE SHIFT.
- #define **SDHC_CMD_CMDTYPE_MASK** (0xC0U)
SDHC COMMAND COMMAND TYPE MASK.
- #define **SDHC_CMD_CMDINDEX_LSF** (8U)
SDHC COMMAND COMMAND INDEX SHIFT.
- #define **SDHC_CMD_CMDINDEX_MASK** (0x3F)
SDHC COMMAND COMMAND INDEX MASK.
- #define **SDHC_RESPONSE** (0x10U)
SDHC RESPONSE REG.
- #define **SDHC_BUFFER** (0x20U)

- #define **SDHC_PRESENT_STATE** (0x24U)
 SDHC PRESENT STATE REG.
- #define **SDHC_PRST_CMD_INHIBIT** (0x1U)
 SDHC PRESENT STATE CMD INHIBIT BIT.
- #define **SDHC_PRST_DATA_INHIBIT** (0x1 << 1)
 SDHC PRESENT STATE DATA INHIBIT BIT.
- #define **SDHC_PRST_DLA** (0x1 << 2)
 SDHC PRESENT STATE DATA LINE ACTIVE BIT.
- #define **SDHC_PRST_RETUNE_REQ** (0x1 << 3)
 SDHC PRESENT STATE RETUNE REQUEST BIT.
- #define **SDHC_PRST_WR_TRANS_A** (0x1 << 8)
 SDHC PRESENT STATE WRITE TRANSFER ACTIVE BIT.
- #define **SDHC_PRST_RD_TRANS_A** (0x1 << 9)
 SDHC PRESENT STATE READ TRANSFER ACTIVE BIT.
- #define **SDHC_PRST_BUFF_WR** (0x1 << 10)
 SDHC PRESENT STATE BUFFER WRITE ENABLE BIT.
- #define **SDHC_PRST_BUFF_RD** (0x1 << 11)
 SDHC PRESENT STATE BUFFER READ ENABLE BIT.
- #define **SDHC_PRST_CARD_INSERTED** (0x1 << 16)
 SDHC PRESENT STATE CARD INSERTED BIT.
- #define **SDHC_PRST_CSS** (0x1 << 17)
 SDHC PRESENT STATE CARD STATE STABLE BIT.
- #define **SDHC_PRST_CD_LVL** (0x1 << 18)
 SDHC PRESENT STATE CARD DETECT PIN LEVEL BIT.
- #define **SDHC_PRST_WP_LVL** (0x1 << 19)
 SDHC PRESENT STATE WRITE PROTECT PIN LEVEL BIT.
- #define **SDHC_PRST_DLSL_0_3_LSF** (20U)
 SDHC PRESENT STATE DAT[3:0] LINE LEVEL SHIFT.
- #define **SDHC_PRST_DLSL_0_3_MASK** (0x0F000000U)
 SDHC PRESENT STATE DAT[3:0] LINE LEVEL MASK.
- #define **SDHC_PRST_CMD_LVL** (0x1 << 24)
 SDHC PRESENT STATE CMD LINE LEVEL BIT.
- #define **SDHC_HOST_CONTROL1** (0x28U)
 SDHC HOST CONTROL1 REG.
- #define **SDHC_CTRL_LED** (0x01U)
 SDHC HOST CONTROL1 LED CONTROL BIT.
- #define **SDHC_CTRL_4BIT** (0x02U)
 SDHC HOST CONTROL1 DATA TRANSFER WIDTH BIT.
- #define **SDHC_CTRL_HISPD** (0x04U)
 SDHC HOST CONTROL1 HIGH SPEED ENABLE BIT.
- #define **SDHC_CTRL_DMA_LSF** (0x3U)
 SDHC HOST CONTROL1 DMA SELECT SHIFT.
- #define **SDHC_CTRL_DMA_MASK** (0x18U)
 SDHC HOST CONTROL1 DMA SELECT MASK.
- #define **SDHC_CTRL_DMA_SDMA** (0x0U)
 SDHC HOST CONTROL1 DMA SELECT SDMA.
- #define **SDHC_CTRL_DMA_ADMA32** (0x2U)
 SDHC HOST CONTROL1 DMA SELECT ADMA32.
- #define **SDHC_CTRL_DMA_ADMA64** (0x3U)
 SDHC HOST CONTROL1 DMA SELECT ADMA64.

SDHC Standard Definition

- #define **SDHC_CTRL_8BIT** (0x20U)
SDHC HOST CONTROL1 EXTENDED DATA TRANSFER WIDTH BIT.
- #define **SDHC_CTRL_CD_TEST_LVL** (0x40U)
SDHC HOST CONTROL1 CARD DETECT TEST LEVEL BIT.
- #define **SDHC_CTRL_CD_SSELECT** (0x80U)
SDHC HOST CONTROL1 CARD DETECT SIGNAL SELECTION BIT.
- #define **SDHC_POWER_CONTROL** (0x29U)
SDHC POWER CONTROL REG.
- #define **SDHC_POWER_ON** (0x01U)
SDHC POWER CONTROL SD BUS POWER.
- #define **SDHC_POWER_180** (0x0A)
SDHC POWER CONTROL SD BUS POWER 1.8V.
- #define **SDHC_POWER_300** (0x0C)
SDHC POWER CONTROL SD BUS POWER 3.0V.
- #define **SDHC_POWER_330** (0x0E)
SDHC POWER CONTROL SD BUS POWER 3.3V.
- #define **SDHC_BLOCK_GAP_CTRL** (0x2A)
SDHC BLOCK GAP CONTROL REG.
- #define **SDHC_BGCTRL_STPATGAPREQ** (0x01U)
SDHC BLOCK GAP CONTROL STOP AT BLOCK GAP BIT.
- #define **SDHC_BGCTRL_CNTNREQ** (0x02U)
SDHC BLOCK GAP CONTROL CONTINUE REQUEST BIT.
- #define **SDHC_BGCTRL_READWAIT** (0x04U)
SDHC BLOCK GAP CONTROL READ WAIT CONTROL BIT.
- #define **SDHC_BGCTRL_INTRATGAP** (0x08U)
SDHC BLOCK GAP CONTROL INTERRUPT AT BLOCK GAP BIT.
- #define **SDHC_WAKEUP_CONTROL** (0x2B)
SDHC WAKEUP CONTROL REG.
- #define **SDHC_WAKE_ON_INT** (0x01U)
SDHC WAKEUP CONTROL WAKEUP ON CARD INTERRUPT BIT.
- #define **SDHC_WAKE_ON_INSERT** (0x02U)
SDHC WAKEUP CONTROL WAKEUP ON CARD INSERTION BIT.
- #define **SDHC_WAKE_ON_REMOVE** (0x04U)
SDHC WAKEUP CONTROL WAKEUP ON CARD REMOVAL BIT.
- #define **SDHC_CLOCK_CONTROL** (0x2C)
SDHC CLOCK CONTROL REG.
- #define **SDHC_CLK_INCLK_EN** (0x0001U)
SDHC CLOCK CONTROL INTERNAL CLOCK ENABLE BIT.
- #define **SDHC_CLK_INCLK_STB** (0x0002U)
SDHC CLOCK CONTROL INTERNAL CLOCK STABLE BIT.
- #define **SDHC_CLK_SDCLK_EN** (0x0004U)
SDHC CLOCK CONTROL SD CLOCK ENABLE BIT.
- #define **SDHC_CLK_CLKGEN_PRG_SEL** (0x0020U)
SDHC CLOCK CONTROL CLOCK GENERATOR SELECTOR BIT.
- #define **SDHC_CLK_FREQ_U_LSF** (6U)
SDHC CLOCK CONTROL UPPER BITS OF FREQUENCY SELECTOR SHIFT.
- #define **SDHC_CLK_FREQ_U_MASK** (0x00C0U)
SDHC CLOCK CONTROL UPPER BITS OF FREQUENCY SELECTOR MASK.
- #define **SDHC_CLK_FREQ_SEL_LSF** (8U)
SDHC CLOCK CONTROL FREQUENCY SELECTOR SHIFT.
- #define **SDHC_CLK_FREQ_SEL_MASK** (0xFF00U)

- #define **SDHC_TIMEOUT_CONTROL** (0x2E)
 SDHC TIMEOUT CONTROL REG.
- #define **SDHC_SOFTWARE_RESET** (0x2F)
 SDHC SOFTWARE RESET REG.
- #define **SDHC_RESET_ALL** (0x01U)
 SDHC SOFTWARE RESET RESET FOR ALL.
- #define **SDHC_RESET_CMD** (0x02U)
 SDHC SOFTWARE RESET RESET FOR CMD LINE.
- #define **SDHC_RESET_DATA** (0x04U)
 SDHC SOFTWARE RESET RESET FOR DATA LINE.
- #define **SDHC_INT_STATUS** (0x30U)
 SDHC NORMAL INTERRUPT STATUS REG.
- #define **SDHC_INT_ENABLE** (0x34U)
 SDHC NORMAL INTERRUPT STATUS ENABLE REG.
- #define **SDHC_SIGNAL_ENABLE** (0x38U)
 SDHC NORMAL INTERRUPT SIGNAL REG.
- #define **SDHC_INT_CMD_DONE** (0x1U << 0)
 SDHC NORMAL INTERRUPT CMD COMPLETE EVENT BIT.
- #define **SDHC_INT_TRANSFER_DONE** (0x1U << 1)
 SDHC NORMAL INTERRUPT TRANSFER COMPLETE EVENT BIT.
- #define **SDHC_INT_BLKGAP_EVENT** (0x1U << 2)
 SDHC NORMAL INTERRUPT BLOCK GAP EVENT BIT.
- #define **SDHC_INT_DMA** (0x1U << 3)
 SDHC NORMAL INTERRUPT DMA EVENT BIT.
- #define **SDHC_INT_WBUF_READY** (0x1U << 4)
 SDHC NORMAL INTERRUPT WRITE BUFFER READY EVENT BIT.
- #define **SDHC_INT_RBUF_READY** (0x1U << 5)
 SDHC NORMAL INTERRUPT READ BUFFER READY EVENT BIT.
- #define **SDHC_INT_CARD_INSERT** (0x1U << 6)
 SDHC NORMAL INTERRUPT CARD INSERTION EVENT BIT.
- #define **SDHC_INT_CARD_REMOVE** (0x1U << 7)
 SDHC NORMAL INTERRUPT CARD REMOVAL EVENT BIT.
- #define **SDHC_INT_CARD_INTR** (0x1U << 8)
 SDHC NORMAL INTERRUPT CARD INTERRUPT BIT.
- #define **SDHC_INT_INT_A** (0x1U << 9)
 SDHC NORMAL INTERRUPT INT_A EVENT BIT.
- #define **SDHC_INT_INT_B** (0x1U << 10)
 SDHC NORMAL INTERRUPT INT_B EVENT BIT.
- #define **SDHC_INT_INT_C** (0x1U << 11)
 SDHC NORMAL INTERRUPT INT_C EVENT BIT.
- #define **SDHC_INT_RETUNING** (0x1U << 12)
 SDHC NORMAL INTERRUPT RETUNING EVENT BIT.
- #define **SDHC_INT_ERROR_INTR** (0x1U << 15)
 SDHC NORMAL INTERRUPT ERROR INTERRUPT BIT.
- #define **SDHC_INT_E_CMD_TIMEOUT** (0x1U << 16)
 SDHC NORMAL INTERRUPT CMD TIMEOUT ERROR BIT.
- #define **SDHC_INT_E_CMD_CRC** (0x1U << 17)
 SDHC NORMAL INTERRUPT CMD CRC ERROR BIT.
- #define **SDHC_INT_E_CMD_END_BIT** (0x1U << 18)
 SDHC NORMAL INTERRUPT CMD INDEX ERROR BIT.

SDHC Standard Definition

- #define **SDHC_INT_E_CMD_INDEX** (0x1U << 19)
SDHC NORMAL INTERRUPT CMD END BIT ERROR BIT.
- #define **SDHC_INT_E_DATA_TIMEOUT** (0x1U << 20)
SDHC NORMAL INTERRUPT DATA TIMEOUT ERROR BIT.
- #define **SDHC_INT_E_DATA_CRC** (0x1U << 21)
SDHC NORMAL INTERRUPT DATA CRC ERROR BIT.
- #define **SDHC_INT_E_DATA_END_BIT** (0x1U << 22)
SDHC NORMAL INTERRUPT DATA END BIT ERROR BIT.
- #define **SDHC_INT_E_CUR_LIMIT** (0x1U << 23)
SDHC NORMAL INTERRUPT CURRENT LIMIT ERROR BIT.
- #define **SDHC_INT_E_AUTOCMD12** (0x1U << 24)
SDHC NORMAL INTERRUPT AUTO CMD12 ERROR BIT.
- #define **SDHC_INT_E_ADMA** (0x1U << 25)
SDHC NORMAL INTERRUPT ADMA ERROR BIT.
- #define **SDHC_INT_E_TUNING** (0x1U << 26)
SDHC NORMAL INTERRUPT TUNING ERROR BIT.
- #define **SDHC_ACMD12_ERROR** (0x3CU)
SDHC AUTO CMD12 ERROR REG.
- #define **SDHC_HOST_CONTROL2** (0x3EU)
SDHC HOST CONTROL2 REG.
- #define **SDHC_CTRL2_UHS_MASK** (0x0007U)
SDHC HOST CONTROL2 UHS MODE MASK.
- #define **SDHC_CTRL2_UHS_SDR12** (0x0000U)
SDHC HOST CONTROL2 UHS-I SDR12.
- #define **SDHC_CTRL2_UHS_SDR25** (0x0001U)
SDHC HOST CONTROL2 UHS-I SDR25.
- #define **SDHC_CTRL2_UHS_SDR50** (0x0002U)
SDHC HOST CONTROL2 UHS-I SDR50.
- #define **SDHC_CTRL2_UHS_SDR104** (0x0003U)
SDHC HOST CONTROL2 UHS-I SDR104.
- #define **SDHC_CTRL2_UHS_DDR50** (0x0004U)
SDHC HOST CONTROL2 UHS-I DDR50.
- #define **SDHC_CTRL2_HS_SDR200** (0x0005U)
SDHC HOST CONTROL2 HS SDR2000.
- #define **SDHC_CTRL2_VDD_180** (0x0008U)
SDHC HOST CONTROL2 1.8V SINGALING ENABLE.
- #define **SDHC_CTRL2_DRV_TYPE_MASK** (0x0030U)
SDHC HOST CONTROL2 DRIVE MASK.
- #define **SDHC_CTRL2_DRV_TYPE_B** (0x0000U)
SDHC HOST CONTROL2 DRIVE TYPE B.
- #define **SDHC_CTRL2_DRV_TYPE_A** (0x0010U)
SDHC HOST CONTROL2 DRIVE TYPE A.
- #define **SDHC_CTRL2_DRV_TYPE_C** (0x0020U)
SDHC HOST CONTROL2 DRIVE TYPE C.
- #define **SDHC_CTRL2_DRV_TYPE_D** (0x0030U)
SDHC HOST CONTROL2 DRIVE TYPE D.
- #define **SDHC_CTRL2_EXEC_TUNING** (0x0040U)
SDHC HOST CONTROL2 EXECUTE TUNING.
- #define **SDHC_CTRL2_TUNED_CLK** (0x0080U)
SDHC HOST CONTROL2 SAMPLING CLOCK SELECT.
- #define **SDHC_CTRL2_ASNYC_INTR_EN** (0x4000U)

- #define **SDHC_CTRL2_PRESET_VAL_EN** (0x8000U)
SDHC HOST CONTROL2 PRESET VALUE ENABLE.
- #define **SDHC_HOST_CAPABILITIES** (0x40U)
SDHC CAPABILITIES REG.
- #define **SDHC_HCAP_TOCLKFREQ_MASK** (0x00000003F)
SDHC CAPABILITIES TIMEOUT CLOCK FREQUENCY.
- #define **SDHC_HCAP_TOCKLUINT_MHZ** (0x000000080U)
SDHC CAPABILITIES TIMEOUT CLOCK UNIT.
- #define **SDHC_HCAP_CLK_BASE_MASK** (0x00003F00U)
SDHC CAPABILITIES BASE CLOCK FREQUENCY FOR SD CLOCK MASK.
- #define **SDHC_HCAP_MAX_BLK_LSF** (16U)
SDHC CAPABILITIES MAX BLOCK LENGTH SHIFT.
- #define **SDHC_HCAP_MAX_BLK_MASK** (0x00030000U)
SDHC CAPABILITIES MAX BLOCK LENGTH MASK.
- #define **SDHC_HCAP_MAXBLK_512** (0x0U)
SDHC CAPABILITIES MAX BLOCK LENGTH 512B.
- #define **SDHC_HCAP_MAXBLK_1024** (0x1U)
SDHC CAPABILITIES MAX BLOCK LENGTH 1024B.
- #define **SDHC_HCAP_MAXBLK_2048** (0x2U)
SDHC CAPABILITIES MAX BLOCK LENGTH 2048B.
- #define **SDHC_HCAP_SUPPORT_8BIT** (0x00040000U)
SDHC CAPABILITIES SUPPORT 8 BIT.
- #define **SDHC_HCAP_SUPPORT_ADMA2** (0x00080000U)
SDHC CAPABILITIES SUPPORT ADMA2.
- #define **SDHC_HCAP_SUPPORT_ADMA1** (0x00100000U)
SDHC CAPABILITIES SUPPORT ADMA1.
- #define **SDHC_HCAP_SUPPORT_HISPD** (0x00200000U)
SDHC CAPABILITIES SUPPORT HIGH SPEED.
- #define **SDHC_HCAP_SUPPORT_SDMA** (0x00400000U)
SDHC CAPABILITIES SUPPORT SDMA.
- #define **SDHC_HCAP_SUPPORT_SUSPEND** (0x00800000U)
SDHC CAPABILITIES SUPPORT SUSPEND RESUME.
- #define **SDHC_HCAP_SUPPORT_V330** (0x01000000U)
SDHC CAPABILITIES SUPPORT 3.3V.
- #define **SDHC_HCAP_SUPPORT_V300** (0x02000000U)
SDHC CAPABILITIES SUPPORT 3.0V.
- #define **SDHC_HCAP_SUPPORT_V180** (0x04000000U)
SDHC CAPABILITIES SUPPORT 1.8V.
- #define **SDHC_HCAP_SUPPORT_64BIT** (0x10000000U)
SDHC CAPABILITIES SUPPORT 64-BIT.
- #define **SDHC_HCAP_SUPPORT_ASYNC** (0x20000000U)
SDHC CAPABILITIES SUPPORT ASYNC INTERRUPT.
- #define **SDHC_HCAP_SLOT_TYPE_LSF** (30U)
SDHC CAPABILITIES SLOT TYPE SHIFT.
- #define **SDHC_HCAP_SLOT_TYPE_MASK** (0xC0000000U)
SDHC CAPABILITIES SLOT TYPE MASK.
- #define **SDHC_HCAP_SLOT_REMOVABLE** (0x0U)
SDHC CAPABILITIES SLOT TYPE REMOVABLE.
- #define **SDHC_HCAP_SLOT_EMBEDDED** (0x1U)
SDHC CAPABILITIES SLOT TYPE EMBEDDED SLOT FOR ONE DEVICE.

SDHC Standard Definition

- #define **SDHC_HCAP_SLOT_SHARED** (0x2U)
SDHC CAPABILITIES SLOT TYPE SHARED BUS SLOT.
- #define **SDHC_HOST_CAPABILITIES_1** (0x44U)
SDHC CAPABILITIES1 REG.
- #define **SDHC_HCAP_SUPPORT_SDR50** (0x00000001U)
SDHC CAPABILITIES1 SUPPORT SDR50.
- #define **SDHC_HCAP_SUPPORT_SDR104** (0x00000002U)
SDHC CAPABILITIES1 SUPPORT SDR104.
- #define **SDHC_HCAP_SUPPORT_DDR50** (0x00000004U)
SDHC CAPABILITIES1 SUPPORT DDR50.
- #define **SDHC_HCAP_DRIVER_TYPE_A** (0x00000010U)
SDHC CAPABILITIES1 SUPPORT DRIVER TYPE A.
- #define **SDHC_HCAP_DRIVER_TYPE_C** (0x00000020U)
SDHC CAPABILITIES1 SUPPORT DRIVER TYPE C.
- #define **SDHC_HCAP_DRIVER_TYPE_D** (0x00000040U)
SDHC CAPABILITIES1 SUPPORT DRIVER TYPE D.
- #define **SDHC_HCAP_RT_TMCNT_LSF** (8U)
SDHC CAPABILITIES1 TIMER COUNT FOR RETUNING SHIFT.
- #define **SDHC_HCAP_RT_TMCNT_MASK** (0x00000F00U)
SDHC CAPABILITIES1 TIMER COUNT FOR RETUNING MASK.
- #define **SDHC_HCAP_USE_SDR50_TUNE** (0x00002000U)
SDHC CAPABILITIES1 USE TUNING FOR SDR50.
- #define **SDHC_HCAP_RT_MODE_LSF** (14U)
SDHC CAPABILITIES1 RETUNE MODE SHIFT.
- #define **SDHC_HCAP_RT_MODE_MASK** (0x0000C000U)
SDHC CAPABILITIES1 RETUNE MODE MASK.
- #define **SDHC_HCAP_CLK_MUL_LSF** (16U)
SDHC CAPABILITIES1 CLOCK MULTIPLIER SHIFT.
- #define **SDHC_HCAP_CLK_MUL_MASK** (0x0FF0000U)
SDHC CAPABILITIES1 CLOCK MULTIPLIER MASK.
- #define **SDHC_MAX_CURRENT** (0x48U)
SDHC MAX CURRENT REG.
- #define **SDHC_MC_330_LSF** (0U)
SDHC MAX CURRENT MAXIMUM CURRENT FOR 3.3V SHIFT.
- #define **SDHC_MC_330_MASK** (0x0000FF)
SDHC MAX CURRENT MAXIMUM CURRENT FOR 3.3V MASK.
- #define **SDHC_MC_300_LSF** (8U)
SDHC MAX CURRENT MAXIMUM CURRENT FOR 3.0V SHIFT.
- #define **SDHC_MC_300_MASK** (0x0FF00U)
SDHC MAX CURRENT MAXIMUM CURRENT FOR 3.0V MASK.
- #define **SDHC_MC_180_LSF** (16U)
SDHC MAX CURRENT MAXIMUM CURRENT FOR 1.8V SHIFT.
- #define **SDHC_MC_180_MASK** (0xFF000U)
SDHC MAX CURRENT MAXIMUM CURRENT FOR 1.8V MASK.
- #define **SDHC_FRC_EVENT_AUTOCMD** (0x50U)
SDHC FORCE EVENT FOR AUTOCMD REG.
- #define **SDHC_FEA_E_NO_ACMD12_EXEC** (0x0001U)
SDHC FORCE EVENT AUTO CMD12 NOT EXECUTED.
- #define **SDHC_FEA_E_ACMD_TIMEOUT** (0x002U)
SDHC FORCE EVENT AUTO CMD TIMEOUT ERROR.
- #define **SDHC_FEA_E_ACMD_CRC** (0x0004U)

- #define SDHC_FEA_E_ACMD_END (0x0008U)
SDHC FORCE EVENT AUTO CMD END BIT ERROR.
- #define SDHC_FEA_E_ACMD_INDEX (0x0010U)
SDHC FORCE EVENT AUTO CMD INDEX ERROR.
- #define SDHC_FEA_E_CMD_NOT_BY_ACMD12 (0x0080U)
SDHC FORCE EVENT AUTO CMD NOT ISSUED ERROR.
- #define SDHC_FRC_EVENT_ERROR_INTR (0x52U)
SDHC FORCE EVENT FOR ERROR REG.
- #define SDHC_FEI_E_CMD_TIMEOUT (0x0001U)
SDHC FORCE CMD TIMEOUT ERROR.
- #define SDHC_FEI_E_CMD_CRC (0x0002U)
SDHC FORCE CMD CRC ERROR.
- #define SDHC_FEI_E_CMD_END_BIT (0x0004U)
SDHC FORCE CMD END BIT ERROR.
- #define SDHC_FEI_E_DATA_TIMEOUT (0x0008U)
SDHC FORCE DATA TIMEOUT ERROR.
- #define SDHC_FEI_E_DATA_CRC (0x0010U)
SDHC FORCE DATA CRC ERROR.
- #define SDHC_FEI_E_DATA_END_BIT (0x0020U)
SDHC FORCE DATA END BIT ERROR.
- #define SDHC_FEI_E_CURRENT_LIMIT (0x0040U)
SDHC FORCE CURRENT LIMIT ERROR.
- #define SDHC_FEI_E_AUTO_CMD (0x0080U)
SDHC FORCE AUTOCMD ERROR.
- #define SDHC_FEI_E_ADMA (0x0100U)
SDHC FORCE ADMA ERROR.
- #define SDHC_ADMA_ERROR (0x54U)
SDHC ADMA ERROR REG.
- #define SDHC_ADMA_ADDRESS (0x58U)
SDHC ADMA ADDRESS REG.
- #define SDHC_SLOT_INT_STATUS (0xFCU)
SDHC SLOT INTERRUPT STATUS REG.
- #define SDHC_HOST_VERSION (0xFEU)
SDHC HOST CONTROLLER VERSION REG.
- #define SDHC_VENDOR_VER_LSF (8U)
SDHC HOST CONTROLLER VERSION VENDOR VERSION SHIFT.
- #define SDHC_VENDOR_VER_MASK (0xFF00U)
SDHC HOST CONTROLLER VERSION VENDOR VERSION MASK.
- #define SDHC_SPEC_VER_LSF (0U)
SDHC HOST CONTROLLER VERSION SPEC VERSION SHIFT.
- #define SDHC_SPEC_VER_MASK (0x00FFU)
SDHC HOST CONTROLLER VERSION SPEC VERSION MASK.
- #define SDHC_SPEC_100 (0U)
SDHC HOST CONTROLLER VERSION SPEC VERSION 1.00.
- #define SDHC_SPEC_200 (1U)
SDHC HOST CONTROLLER VERSION SPEC VERSION 2.00.
- #define SDHC_SPEC_300 (2U)
SDHC HOST CONTROLLER VERSION SPEC VERSION 3.00.

SDHC Card Related Standard Definition

22.5 SDHC Card Related Standard Definition

This chapter describes the card standard definition.

Macros

- #define **SDMMC_CARD_BUSY** ((uint32_t) 1 << 31)
card initialization complete
- #define **SD_SCR_BUS_WIDTHS_1BIT** (1 << 0)
card supports 1 bit mode
- #define **SD_SCR_BUS_WIDTHS_4BIT** (1 << 2)
card supports 4 bit mode
- #define **SD_CCC_BASIC** (1 << 0)
Card command class 0.
- #define **SD_CCC_BLOCK_READ** (1 << 2)
Card command class 2.
- #define **SD_CCC_BLOCK_WRITE** (1 << 4)
Card command class 4.
- #define **SD_CCC_ERASE** (1 << 5)
Card command class 5.
- #define **SD_CCC_WRITE_PROTECTION** (1 << 6)
Card command class 6.
- #define **SD_CCC_LOCK_CARD** (1 << 7)
Card command class 7.
- #define **SD_CCC_APP_SPEC** (1 << 8)
Card command class 8.
- #define **SD_CCC_IO_MODE** (1 << 9)
Card command class 9.
- #define **SD_CCC_SWITCH** (1 << 10)
Card command class 10.
- #define **SD_OCR_CCS** (1 << 30)
card capacity status
- #define **SD_OCR_HCS** (1 << 30)
card capacity status
- #define **SD_OCR_XPC** (1 << 28)
SDXC power control.
- #define **SD_OCR_S18R** (1 << 24)
switch to 1.8V request
- #define **SD_OCR_S18A SD_OCR_S18R**
switch to 1.8V accepted
- #define **SD_HIGHSPEED_BUSY** (0x00020000U)
SD card high speed busy status bit in CMD6 response.
- #define **SD_HIGHSPEED_SUPPORTED** (0x00020000U)
SD card high speed support bit in CMD6 response.
- #define **SD_OCR_VDD_27_28** (1 << 15)
VDD 2.7-2.8.
- #define **SD_OCR_VDD_28_29** (1 << 16)
VDD 2.8-2.9.
- #define **SD_OCR_VDD_29_30** (1 << 17)
VDD 2.9-3.0.
- #define **SD_OCR_VDD_30_31** (1 << 18)

- #define **SD_OCR_VDD_31_32** (1 << 19)
VDD 3.0-3.1.
- #define **SD_OCR_VDD_32_33** (1 << 20)
VDD 3.1-3.2.
- #define **SD_OCR_VDD_33_34** (1 << 21)
VDD 3.2-3.3.
- #define **SD_OCR_VDD_34_35** (1 << 22)
VDD 3.3-3.4.
- #define **SD_OCR_VDD_35_36** (1 << 23)
VDD 3.4-3.5.
- #define **SDMMC_R1_OUT_OF_RANGE** ((uint32_t) 1 << 31)
R1: out of range status bit.
- #define **SDMMC_R1_ADDRESS_ERROR** (1 << 30)
R1: address error status bit.
- #define **SDMMC_R1_BLOCK_LEN_ERROR** (1 << 29)
R1: block length error status bit.
- #define **SDMMC_R1_ERASE_SEQ_ERROR** (1 << 28)
R1: erase sequence error status bit.
- #define **SDMMC_R1_ERASE_PARAM** (1 << 27)
R1: erase parameter error status bit.
- #define **SDMMC_R1_WP_VIOLATION** (1 << 26)
R1: write protection violation status bit.
- #define **SDMMC_R1_CARD_IS_LOCKED** (1 << 25)
R1: card locked status bit.
- #define **SDMMC_R1_LOCK_UNLOCK_FAILED** (1 << 24)
R1: lock/unlock error status bit.
- #define **SDMMC_R1_COM_CRC_ERROR** (1 << 23)
R1: CRC error status bit.
- #define **SDMMC_R1_ILLEGAL_COMMAND** (1 << 22)
R1: illegal command status bit.
- #define **SDMMC_R1_CARD_ECC_FAILED** (1 << 21)
R1: card ecc error status bit.
- #define **SDMMC_R1_CC_ERROR** (1 << 20)
R1: internal card controller status bit.
- #define **SDMMC_R1_ERROR** (1 << 19)
R1: a general or an unknown error status bit.
- #define **SDMMC_R1_CID_CSD_OVERWRITE** (1 << 16)
R1: cid/csd overwrite status bit.
- #define **SDMMC_R1_WP_ERASE_SKIP** (1 << 15)
R1: write protection erase skip status bit.
- #define **SDMMC_R1_CARD_ECC_DISABLED** (1 << 14)
R1: card ecc disabled status bit.
- #define **SDMMC_R1_ERASE_RESET** (1 << 13)
R1: erase reset status bit.
- #define **SDMMC_R1_STATUS**(x) ((uint32_t)(x) & 0xFFFFE000U)
R1: status.
- #define **SDMMC_R1_READY_FOR_DATA** (1 << 8)
R1: ready for data status bit.
- #define **SDMMC_R1_SWITCH_ERROR** (1 << 7)
R1: switch error status bit.

SDHC Card Related Standard Definition

- #define **SDMMC_R1_APP_CMD** (1 << 5)
R1: application command enabled status bit.
- #define **SDMMC_R1_AKE_SEQ_ERROR** (1 << 3)
R1: error in the sequence of the authentication process.
- #define **SDMMC_R1_ERROR_BITS**(x)
Check error card status.
- #define **SDMMC_R1_CURRENT_STATE**(x) (((x) & 0x00001E00U) >> 9)
R1: current state.
- #define **SDMMC_R1_STATE_IDLE** (0U)
R1: current state: idle.
- #define **SDMMC_R1_STATE_READY** (1U)
R1: current state: ready.
- #define **SDMMC_R1_STATE_IDENT** (2U)
R1: current state: ident.
- #define **SDMMC_R1_STATE_STBY** (3U)
R1: current state: stby.
- #define **SDMMC_R1_STATE_TRAN** (4U)
R1: current state: tran.
- #define **SDMMC_R1_STATE_DATA** (5U)
R1: current state: data.
- #define **SDMMC_R1_STATE_RCV** (6U)
R1: current state: rcv.
- #define **SDMMC_R1_STATE_PRG** (7U)
R1: current state: prg.
- #define **SDMMC_R1_STATE_DIS** (8U)
R1: current state: dis.
- #define **SDMMC_SD_VERSION_1_0** (1 << 0)
SD card version 1.0.
- #define **SDMMC_SD_VERSION_1_1** (1 << 1)
SD card version 1.1.
- #define **SDMMC_SD_VERSION_2_0** (1 << 2)
SD card version 2.0.
- #define **SDMMC_SD_VERSION_3_0** (1 << 3)
SD card version 3.0.

Enumerations

- enum `mmc_cmd_t` {

 `kMmcSetRelativeAddr` = 3,

 `kMmcSleepAwake` = 5,

 `kMmcSwitch` = 6,

 `kMmcSendExtCsd` = 8,

 `kMmcReadDataUntilStop` = 11,

 `kMmcBusTestRead` = 14,

 `kMmcWriteDataUntilStop` = 20,

 `kMmcProgramCid` = 26,

 `kMmcEraseGroupStart` = 35,

 `kMmcEraseGroupEnd` = 36,

 `kMmcFastIo` = 39,

 `kMmcGoIrqState` = 40 }
- enum `sdmmc_cmd_t` {

 `kGoIdleState` = 0,

 `kSendOpCond` = 1,

 `kAllSendCid` = 2,

 `kSetDsr` = 4,

 `kSelectCard` = 7,

 `kSendCsd` = 9,

 `kSendCid` = 10,

 `kStopTransmission` = 12,

 `kSendStatus` = 13,

 `kGoInactiveState` = 15,

 `kSetBlockLen` = 16,

 `kReadSingleBlock` = 17,

 `kReadMultipleBlock` = 18,

 `kSendTuningBlock` = 19,

 `kSetBlockCount` = 23,

 `kWriteBlock` = 24,

 `kWriteMultipleBlock` = 25,

 `kProgramCsd` = 27,

 `kSetWriteProt` = 28,

 `kClrWriteProt` = 29,

 `kSendWriteProt` = 30,

 `kErase` = 38,

 `kLockUnlock` = 42,

 `kAppCmd` = 55,

 `kGenCmd` = 56 }
- enum `sd_cmd_t` {

SDHC Card Related Standard Definition

```
kSdSendRelativeAddr = 3,  
kSdSwitch = 6,  
kSdSendIfCond = 8,  
kSdVoltageSwitch = 11,  
kSdSpeedClassControl = 20,  
kSdEraseWrBlkStart = 32,  
kSdEraseWrBlkEnd = 33 }  
• enum sd_acmd_t {  
    kSdAppSetBusWidth = 6,  
    kSdAppStatus = 13,  
    kSdAppSendNumWrBlocks = 22,  
    kSdAppSetWrBlkEraseCount = 23,  
    kSdAppSendOpCond = 41,  
    kSdAppSetClrCardDetect = 42,  
    kSdAppSendScr = 51 }  
• enum sd_buswidth_t {  
    kSdBusWidth1Bit = 0,  
    kSdBusWidth4Bit = 2 }  
• enum sd_switch_mode_t {  
    kSdSwitchCheck = 0,  
    kSdSwitchSet = 1 }
```

22.5.1 Enumeration Type Documentation

22.5.1.1 enum mmc_cmd_t

Enumerator

kMmcSetRelativeAddr ac [31:16] RCA R1
kMmcSleepAwake ac [31:16] RCA R1b [15] flag
kMmcSwitch ac [31:16] RCA R1b
kMmcSendExtCsd adtc R1
kMmcReadDataUntilStop adtc [31:0] data R1 address
kMmcBusTestRead adtc R1
kMmcWriteDataUntilStop ac [31:0] data R1 address
kMmcProgramCid adtc R1
kMmcEraseGroupStart ac [31:0] data R1 address
kMmcEraseGroupEnd ac [31:0] data R1 address
kMmcFastIo ac R4
kMmcGoIrqState bcr R5

22.5.1.2 enum sdmmc_cmd_t

Enumerator

```

kGoIdleState bc
kSendOpCond bcr [31:0] OCR R3
kAllSendCid bcr R2
kSetDsr bc [31:16] RCA
kSelectCard ac [31:16] RCA R1b
kSendCsd ac [31:16] RCA R2
kSendCid ac [31:16] RCA R2
kStopTransmission ac [31:16] RCA R1b
kSendStatus ac [31:16] RCA R1
kGoInactiveState ac [31:16] RCA
kSetBlockLen ac [31:0] block R1 length
kReadSingleBlock adtc [31:0] data R1 address
kReadMultipleBlock adtc [31:0] data R1 address
kSendTuningBlock adtc [31:0] all R1 zero
kSetBlockCount ac [31:0] block R1 count
kWriteBlock adtc [31:0] data R1 address
kWriteMultipleBlock adtc [31:0] data R1 address
kProgramCsd adtc R1
kSetWriteProt ac [31:0] data R1b address
kClrWriteProt ac [31:0] data R1b address
kSendWriteProt adtc [31:0] write R1b protect data address
kErase ac R1
kLockUnlock adtc all zero R1
kAppCmd ac [31:16] RCA R1
kGenCmd adtc [0] RD/WR R1

```

22.5.1.3 enum sd_cmd_t

Enumerator

```

kSdSendRelativeAddr bcr R6
kSdSwitch adtc [31] mode R1 [15:12] func group 4: current limit [11:8] func group 3: drive strength
[7:4] func group 2: command system [3:0] func group 1: access mode
kSdSendIfCond bcr [11:8] supply R7 voltage [7:0] check pattern
kSdVoltageSwitch ac R1
kSdSpeedClassControl ac [31:28] speed R1b class control
kSdEraseWrBlkStart ac [31:0] data R1 address
kSdEraseWrBlkEnd ac [31:0] data R1 address

```

SDHC Card Related Standard Definition

22.5.1.4 enum sd_acmd_t

Enumerator

kSdAppSetBusWdith ac [1:0] bus R1 width
kSdAppStatus adtc R1
kSdAppSendNumWrBlocks adtc R1
kSdAppSetWrBlkEraseCount ac [22:0] number R1 of blocks
kSdAppSendOpCond bcr [30] HCS R3 [28] XPC [24] S18R [23:0] VDD voltage window
kSdAppSetClrCardDetect ac [0] set cd R1
kSdAppSendScr adtc R1

22.5.1.5 enum sd_buswidth_t

Enumerator

kSdBusWidth1Bit SD data bus width 1-bit mode.
kSdBusWidth4Bit SD data bus width 4-bit mode.

22.5.1.6 enum sd_switch_mode_t

Enumerator

kSdSwitchCheck SD switch mode 0: check function.
kSdSwitchSet SD switch mode 1: set function.

22.6 SDHC Card Definition

This chapter describes the programming interface of the card data definition for FSL SD host controller.

Data Structures

- struct [sdhc_card_t](#)
SDHC Card Structure. [More...](#)

Macros

- #define [FSL_SDHC_CARD_DEFAULT_BLOCK_SIZE](#) (512)
Default block size.

Enumerations

- enum [sdhc_card_type_t](#) {

 kCardTypeUnknown = 1,

 kCardTypeSd,

 kCardTypeMmc,

 kCardTypeSdio }

22.6.1 Data Structure Documentation

22.6.1.1 struct [sdhc_card_t](#)

Defines the card structure including the necessary fields to identify and describe the card.

Data Fields

- [uint32_t hostInstance](#)
Host instance id.
- [sdhc_card_type_t cardType](#)
Card type.
- [uint32_t rca](#)
Relative address of the card.
- [uint32_t version](#)
Card version.
- [uint32_t caps](#)
Capability.
- [uint32_t rawCid \[4\]](#)
CID.
- [uint32_t rawCsd \[4\]](#)
CSD.

SDHC Card Definition

- `uint32_t rawScr [2]`
CSD.
- `uint32_t ocr`
OCR.
- `sdcards_cid_t cid`
CID.
- `sdcards_csd_t csd`
CSD.
- `sdcards_scr_t scr`
SCR.
- `uint32_t blockCount`
Card total block number.
- `uint32_t blockSize`
Card block size.

22.6.2 Enumeration Type Documentation

22.6.2.1 enum sdhc_card_type_t

Enumerator

kCardTypeUnknown Unknown card type.

kCardTypeSd SD card type.

kCardTypeMmc MMC card type.

kCardTypeSdio SDIO card type.

22.7 SDHC Card Driver

This chapter describes the programming interface of the card driver for FSL SD host controller.

SDHC CARD DRIVER FUNCTION

- `sdhc_status_t SDCARD_DRV_Init (sdhc_host_t *host, sdhc_card_t *card)`
Initializes the card on a specific host controller.
- `sdhc_status_t SDCARD_DRV_ReadBlocks (sdhc_card_t *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`
Reads blocks from the specific card.
- `sdhc_status_t SDCARD_DRV_WriteBlocks (sdhc_card_t *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`
Writes blocks of data to the specific card.
- `sdhc_status_t SDCARD_DRV_EraseBlocks (sdhc_card_t *card, uint32_t startBlock, uint32_t blockCount)`
Erases blocks of the specific card.
- `bool SDCARD_DRV_CheckReadOnly (sdhc_card_t *card)`
Checks whether the card is write-protected.
- `void SDCARD_DRV_Shutdown (sdhc_card_t *card)`
Deinitializes the card.

22.7.1 Function Documentation

22.7.1.1 `sdhc_status_t SDCARD_DRV_Init (sdhc_host_t * host, sdhc_card_t * card)`

This function initializes the card on a specific SDHC.

Parameters

<code>host</code>	the pointer to the host struct, it is allocated by user
<code>card</code>	the place to store card related information

Returns

`kStatus_SDHC_NoError` on success

22.7.1.2 `sdhc_status_t SDCARD_DRV_ReadBlocks (sdhc_card_t * card, uint8_t * buffer, uint32_t startBlock, uint32_t blockCount)`

This function reads blocks from specific card, with default block size defined by `FSL_SDHC_CARD_DEFAULT_BLOCK_SIZE`.

SDHC Card Driver

Parameters

<i>card</i>	the handle of the card
<i>buffer</i>	the buffer to hold the data read from card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to read

Returns

kStatus_SDHC_NoError on success

22.7.1.3 sdhc_status_t SDCARD_DRV_WriteBlocks (*sdhc_card_t * card, uint8_t * buffer, uint32_t startBlock, uint32_t blockCount*)

This function writes blocks to specific card, with default block size defined by FSL_SDHC_CARD_DEFAULT_BLOCK_SIZE.

Parameters

<i>card</i>	the handle of the card
<i>buffer</i>	the buffer holding the data to be written to the card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to write

Returns

kStatus_SDHC_NoError on success

22.7.1.4 sdhc_status_t SDCARD_DRV_EraseBlocks (*sdhc_card_t * card, uint32_t startBlock, uint32_t blockCount*)

This function erases blocks of a specific card, with default block size defined by the FSL_SDHC_CARD_DEFAULT_BLOCK_SIZE.

Parameters

<i>card</i>	the handle of the card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to erase

Returns

kStatus_SDHC_NoError on success

22.7.1.5 bool SDCARD_DRV_CheckReadOnly (*sdhc_card_t * card*)

This function checks if the card is write-protected via CSD register.

Parameters

<i>card</i>	the specific card
-------------	-------------------

Returns

kStatus_SDHC_NoError on success

22.7.1.6 void SDCARD_DRV_Shutdown (*sdhc_card_t * card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	the specific card
-------------	-------------------

Chapter 23

Soundcard (SND)

The Kinetis SDK provides both HAL and Peripheral drivers for the Soundcard (SND) block of Kinetis devices.

Data Structures

- struct `snd_state_t`
Soundcard status includes the information which the application can see. [More...](#)
- struct `audio_ctrl_operation_t`
The operations of an audio controller, for example SAI, SSI and so on. [More...](#)
- struct `audio_codec_operation_t`
Audio codec operation structure. [More...](#)
- struct `audio_controller_t`
The definition of the audio device which may be a controller. [More...](#)
- struct `audio_codec_t`
The Codec structure. [More...](#)
- struct `audio_buffer_t`
Audio buffer structure. [More...](#)
- struct `sound_card_t`
A sound card includes the audio controller and a Codec. [More...](#)

Macros

- #define `USEDMA` 1
Use DMA mode or interrupt mode.
- #define `AUDIO_CONTROLLER` `AUDIO_CONTROLLER_SAI`
Define audio controller sai.
- #define `AUDIO_CODEC` `AUDIO_CODEC_SGTL5000`
Define audio codec sgtl5000.
- #define `AUDIO_BUFFER_BLOCK_SIZE` 1024
Buffer block size setting.
- #define `AUDIO_BUFFER_BLOCK` 2
Buffer block number setting.
- #define `AUDIO_TX` 1
Audio transfer direction Tx.
- #define `AUDIO_RX` 0
Audio transfer direction Rx.

Enumerations

- enum `snd_status_t` {
 `kStatus_SND_Success` = 0U,
 `kStatus_SND_Fail` = 1U,
 `kStatus_SND_DmaFail` = 2U,
 `kStatus_SND_CtrlFail` = 3U,
 `kStatus_SND_CodecFail` = 4U,
 `kStatus_SND_BufferAllocateFail` = 5U }

Soundcard return status.

Functions

- `snd_status_t SND_TxInit (sound_card_t *card, void *ctrl_config, void *codec_config, ctrl_state_t *state)`
Initializes the Playback Soundcard.
- `snd_status_t SND_RxInit (sound_card_t *card, void *ctrl_config, void *codec_config, ctrl_state_t *state)`
Initializes the record Soundcard.
- `snd_status_t SND_TxDeinit (sound_card_t *card)`
Deinitializes the playback Soundcard.
- `snd_status_t SND_RxDeinit (sound_card_t *card)`
Deinitializes the playback Soundcard.
- `snd_status_t SND_TxConfigDataFormat (sound_card_t *card, ctrl_data_format_t *format)`
Configures the audio data format running in the Soundcard.
- `snd_status_t SND_RxConfigDataFormat (sound_card_t *card, ctrl_data_format_t *format)`
Configures the audio data format running in the Soundcard.
- `uint32_t SND_TxUpdateStatus (sound_card_t *card, uint32_t len)`
Updates the status of the TX.
- `uint32_t SND_RxUpdateStatus (sound_card_t *card, uint32_t len)`
Updates the status of the Rx.
- `void SND_GetStatus (sound_card_t *card, snd_state_t *status)`
Gets the status of the Soundcard.
- `void SND_TxStart (sound_card_t *card)`
Starts the Soundcard Tx process.
- `void SND_RxStart (sound_card_t *card)`
Starts the Soundcard Rx process.
- `static void SND_TxStop (sound_card_t *card)`
Stops the Soundcard Tx process.
- `static void SND_RxStop (sound_card_t *card)`
Stops Soundcard RX process.
- `void SND_WaitEvent (sound_card_t *card)`
Waits for the semaphore of the write/read data from the internal buffer.
- `snd_status_t SND_SetMuteCmd (sound_card_t *card, bool enable)`
Mutes the Soundcard.
- `snd_status_t SND_SetVolume (sound_card_t *card, uint32_t volume)`
Sets the volume of the Soundcard.
- `uint32_t SND_GetVolume (sound_card_t *card)`
Gets the volume of the Soundcard.

23.0.2 Soundcard Driver

Overview

The Soundcard handles both audio controller and audio codec. It also provides an easy way to initialize, configure and handle(read/write) the Soundcard.

Initialization

A Soundcard includes a controller and a Codec (SAI + sg15000). The application needs to prepare the configuration structure for SAI and SGTL5000, and then call the [SND_TxInit\(\)](#) function to finish the initialization.

Configuration

The configuration includes the static configuration and dynamic configuration. Static configuration involves configuring the SAI features. Dynamic configuration involves configuring the audio data format (sample rate and bit depth).

Call diagram

To use the SAI driver follow these steps using the Tx as an example:

1. Initialize the Soundcard by calling the [SND_TxInit\(\)](#) function.
2. Configure the audio data format information of the Soundcard by calling the [SND_TxConfigDataFormat\(\)](#) function.
3. Update the status of the Soundcard after copying into/output from the buffer by calling the [SND_TxUpdateStatus\(\)](#) function.
4. Start the TX/RX by calling the [SND_TxStart\(\)](#) or the [SND_RxStart\(\)](#) function.
5. Stop by calling the [SND_TxStop\(\)](#) or [SND_RxStop\(\)](#) functions.
6. Close the devices by calling the [SND_TxDeinit\(\)](#) function.

This is an example code to initialize and configure the SAI driver in the DMA mode:

```
sound_card_t g_card;
static audio_data_format_t *format;
static sai_user_config_t tx_config;
static sai_state_t tx_state;
static edma_state_t edmaState;
static edma_user_config_t edmaUserConfig;

void snd_card_config(void)
{
    /* SAI configuration */
    OSA_Init();
    /* Configure the play audio format */
    format = (sai_data_format_t *)OSA_MemAllocZero(sizeof(
        sai_data_format_t));
    format->bits = 16;
    format->sample_rate = 96000;
    format->mclk = 384 * format->sample_rate;
    format->mono_stereo = kSaiStereo;

    /* SAI configuration */
```

Data Structure Documentation

```
tx_config.protocol = kSaiBusI2SLeft;
tx_config.channel = 0;
tx_config.slave_master = kSaiMaster;
tx_config.sync_mode = kSaiModeAsync;
tx_config.bclk_source = kSaiBclkSourceMclkDiv;
tx_config.mclk_source = kSaiMclkSourceSysclk;
tx_config.mclk_divide_enable = true;
tx_config.watermark = 4;
g_card.controller.instance = 0;
g_card.controller.fifo_channel = 0;
g_card.controller.ops = &g_sai_ops;
g_card.controller.dma_source = kDmaRequestMux0I2S0Tx;
sgtl_handler_t *codec_handler = (sgtl_handler_t *)OSA_MemAllocZero(sizeof(
sgtl_handler_t));
g_card.codec.handler = codec_handler;
g_card.codec.ops = &g_sgtl_ops;
}

// Initialize the Soundcard
edmaUserConfig.chnArbitration = kEDMACHnArbitrationRoundrobin
;
edmaUserConfig.notHaltOnError = false;
EDMA_DRV_Init(&edmaState, &edmaUserConfig);
#endif
SND_TxInit(&g_card, &tx_config, NULL, &tx_state);

// Configure the audio data format for TX
SND_TxConfigDataFormat(&g_card, format);

// Copy data to SAI buffer
SND_WaitEvent(&g_card);
SND_GetStatus(&g_card, &tx_status);
memcpy(tx_status.input_address, pData, tx_status.size);
SND_TxUpdateStatus(&g_card, tx_status.size);
```

23.1 Data Structure Documentation

23.1.1 struct snd_state_t

This structure is the interface between the driver and the application. The application can get the information where and when to input/output data.

Data Fields

- uint32_t **size**
The size of a block.
- uint32_t **empty_block**
How many blocks are empty.
- uint32_t **full_block**
How many blocks are full.
- uint8_t * **input_address**
The input address.
- uint8_t * **output_address**
The output address.

23.1.2 struct audio_ctrl_operation_t

The operation includes the basic initialize, configure, send, receive and so on.

Data Fields

- `ctrl_status_t(* Ctrl_TxInit)(uint32_t instance, ctrl_config_t *config, ctrl_state_t *state)`
Initializes Tx.
- `ctrl_status_t(* Ctrl_RxInit)(uint32_t instance, ctrl_config_t *config, ctrl_state_t *state)`
Initializes Rx.
- `ctrl_status_t(* Ctrl_TxDeinit)(uint32_t instance)`
Deinitializes Tx.
- `ctrl_status_t(* Ctrl_RxDeinit)(uint32_t instance)`
Deinitializes Rx.
- `ctrl_status_t(* Ctrl_TxConfigDataFormat)(uint32_t instance, ctrl_data_format_t *format)`
Configures Tx audio data format.
- `ctrl_status_t(* Ctrl_RxConfigDataFormat)(uint32_t instance, ctrl_data_format_t *format)`
Configures Rx audio data format.
- `void(* Ctrl_TxStart)(uint32_t instance)`
Used in a start transfer or a resume transfer.
- `void(* Ctrl_RxStart)(uint32_t instance)`
Used in a start receive or a resume receive.
- `void(* Ctrl_TxStop)(uint32_t instance)`
Used to stop transfer.
- `void(* Ctrl_RxStop)(uint32_t instance)`
Used to stop receive.
- `void(* Ctrl_TxRegisterCallback)(uint32_t instance, ctrl_callback_t callback, void *callback_param)`
Registers a tx callback function.
- `void(* Ctrl_RxRegisterCallback)(uint32_t instance, ctrl_callback_t callback, void *callback_param)`
Registers an rx callback function.
- `void(* Ctrl_TxSetIntCmd)(uint32_t instance, bool enable)`
Enable/disable Tx interrupt.
- `void(* Ctrl_RxSetIntCmd)(uint32_t instance, bool enable)`
Enable/disable Rx interrupt.
- `void(* Ctrl_TxSetDmaCmd)(uint32_t instance, bool enable)`
Enable/disable Tx DMA.
- `void(* Ctrl_RxSetDmaCmd)(uint32_t instance, bool enable)`
Enable/disable Rx DMA.
- `uint32_t(* Ctrl_TxGetWatermark)(uint32_t instance)`
Get watermark of T.
- `uint32_t(* Ctrl_RxGetWatermark)(uint32_t instance)`
Get watermark of Rx.
- `uint32_t *(* Ctrl_TxGetFifoAddr)(uint32_t instance, uint32_t fifo_channel)`
Gets Tx FIFO address.
- `uint32_t *(* Ctrl_RxGetFifoAddr)(uint32_t instance, uint32_t fifo_channel)`
Gets Rx FIFO address.
- `uint32_t(* Ctrl_SendData)(uint32_t instance, uint8_t *addr, uint32_t len)`

Data Structure Documentation

- *Sends data function.*
• `uint32_t(* Ctrl_ReceiveData)(uint32_t instance, uint8_t *addr, uint32_t len)`
Receives data.

23.1.2.0.0.48 Field Documentation

- 23.1.2.0.0.48.1 `ctrl_status_t(* audio_ctrl_operation_t::Ctrl_TxInit)(uint32_t instance, ctrl_config_t *config, ctrl_state_t *state)`
- 23.1.2.0.0.48.2 `ctrl_status_t(* audio_ctrl_operation_t::Ctrl_RxInit)(uint32_t instance, ctrl_config_t *config, ctrl_state_t *state)`
- 23.1.2.0.0.48.3 `ctrl_status_t(* audio_ctrl_operation_t::Ctrl_TxDeinit)(uint32_t instance)`
- 23.1.2.0.0.48.4 `ctrl_status_t(* audio_ctrl_operation_t::Ctrl_RxDeinit)(uint32_t instance)`
- 23.1.2.0.0.48.5 `ctrl_status_t(* audio_ctrl_operation_t::Ctrl_TxConfigDataFormat)(uint32_t instance, ctrl_data_format_t *format)`
- 23.1.2.0.0.48.6 `ctrl_status_t(* audio_ctrl_operation_t::Ctrl_RxConfigDataFormat)(uint32_t instance, ctrl_data_format_t *format)`
- 23.1.2.0.0.48.7 `void(* audio_ctrl_operation_t::Ctrl_TxStop)(uint32_t instance)`
- 23.1.2.0.0.48.8 `void(* audio_ctrl_operation_t::Ctrl_RxStop)(uint32_t instance)`
- 23.1.2.0.0.48.9 `void(* audio_ctrl_operation_t::Ctrl_TxRegisterCallback)(uint32_t instance, ctrl_callback_t callback, void *callback_param)`
- 23.1.2.0.0.48.10 `void(* audio_ctrl_operation_t::Ctrl_RxRegisterCallback)(uint32_t instance, ctrl_callback_t callback, void *callback_param)`
- 23.1.2.0.0.48.11 `void(* audio_ctrl_operation_t::Ctrl_TxSetIntCmd)(uint32_t instance, bool enable)`
- 23.1.2.0.0.48.12 `void(* audio_ctrl_operation_t::Ctrl_RxSetIntCmd)(uint32_t instance, bool enable)`
- 23.1.2.0.0.48.13 `void(* audio_ctrl_operation_t::Ctrl_TxSetDmaCmd)(uint32_t instance, bool enable)`
- 23.1.2.0.0.48.14 `void(* audio_ctrl_operation_t::Ctrl_RxSetDmaCmd)(uint32_t instance, bool enable)`
- 23.1.2.0.0.48.15 `uint32_t(* audio_ctrl_operation_t::Ctrl_TxGetWatermark)(uint32_t instance)`
- 23.1.2.0.0.48.16 `uint32_t(* audio_ctrl_operation_t::Ctrl_RxGetWatermark)(uint32_t instance)`

23.1.3 struct audio_codec_operation_t

Data Fields

- `codec_status_t(* Codec_Init)(codec_handler_t *param, codec_init_t *config)`
Codec initialize function.
- `codec_status_t(* Codec_Deinit)(codec_handler_t *param)`

Data Structure Documentation

- *Codec deinitialize function.*
codec_status_t(* [Codec_ConfigDataFormat](#))(codec_handler_t *param, uint32_t mclk, uint32_t sample_rate, uint8_t bits)
Configures data format.
- codec_status_t(* [Codec_SetMuteCmd](#))(codec_handler_t *param, bool enable)
Mute and unmute.
- codec_status_t(* [Codec_SetVolume](#))(codec_handler_t *param, uint32_t volume)
Set volume.
- uint32_t(* [Codec_GetVolume](#))(codec_handler_t *param)
Get volume.

23.1.3.0.0.49 Field Documentation

23.1.3.0.0.49.1 `codec_status_t(* audio_codec_operation_t::Codec_ConfigDataFormat)(codec_handler_t *param, uint32_t mclk, uint32_t sample_rate, uint8_t bits)`

23.1.3.0.0.49.2 `codec_status_t(* audio_codec_operation_t::Codec_SetMuteCmd)(codec_handler_t *param, bool enable)`

23.1.3.0.0.49.3 `codec_status_t(* audio_codec_operation_t::Codec_SetVolume)(codec_handler_t *param, uint32_t volume)`

23.1.3.0.0.49.4 `uint32_t(* audio_codec_operation_t::Codec_GetVolume)(codec_handler_t *param)`

23.1.4 struct audio_controller_t

Data Fields

- [edma_chn_state_t](#) `dma_channel`
Which DMA channel it uses.
- [dma_request_source_t](#) `dma_source`
DMA request source.
- [edma_software_tcd_t](#) `stcd` [[AUDIO_BUFFER_BLOCK](#)+1]
TCDs for eDMA configuration.
- [audio_ctrl_operation_t](#) * `ops`
Operations including initialization, configuration, etc.

23.1.4.0.0.50 Field Documentation**23.1.4.0.0.50.1 edma_software_tcd_t audio_controller_t::stcd[AUDIO_BUFFER_BLOCK+1]****23.1.4.0.0.50.2 audio_ctrl_operation_t* audio_controller_t::ops****23.1.5 struct audio_codec_t****Data Fields**

- void * **handler**
Codec instance.
- **audio_codec_operation_t * ops**
Operations.

23.1.5.0.0.51 Field Documentation**23.1.5.0.0.51.1 audio_codec_operation_t* audio_codec_t::ops****23.1.6 struct audio_buffer_t****Data Fields**

- uint8_t * **buff**
Buffer address.
- uint8_t **blocks**
Block number of the buffer.
- uint16_t **size**
The size of a block.
- uint32_t **requested**
The request data number to transfer.
- uint32_t **queued**
Data which is in buffer, but not processed.
- uint32_t **processed**
Data which is put into the FIFO.
- uint8_t **input_index**
Buffer input block index.
- uint8_t **output_index**
Buffer output block index.
- uint8_t * **input_curbuff**
Buffer input address.
- uint8_t * **output_curbuff**
Buffer output address.
- uint32_t **empty_block**
Empty block number.
- uint32_t **full_block**
Full block numbers.
- uint32_t **fifo_error**
FIFO error numbers.

Data Structure Documentation

- `uint32_t buffer_error`
Buffer error numbers.
- `semaphore_t sem`
Semaphores to control the data flow.
- `bool first_io`
Means the first time the transfer.

23.1.6.0.0.52 Field Documentation

23.1.6.0.0.52.1 `uint8_t audio_buffer_t::blocks`

23.1.6.0.0.52.2 `uint32_t audio_buffer_t::queued`

23.1.6.0.0.52.3 `uint32_t audio_buffer_t::processed`

This is used to judge if the SAI is under run.

23.1.6.0.0.52.4 `uint8_t audio_buffer_t::input_index`

23.1.6.0.0.52.5 `uint8_t audio_buffer_t::output_index`

23.1.6.0.0.52.6 `uint8_t* audio_buffer_t::input_curbuff`

23.1.6.0.0.52.7 `uint8_t* audio_buffer_t::output_curbuff`

23.1.6.0.0.52.8 `uint32_t audio_buffer_t::empty_block`

23.1.6.0.0.52.9 `uint32_t audio_buffer_t::full_block`

23.1.6.0.0.52.10 `uint32_t audio_buffer_t::fifo_error`

23.1.6.0.0.52.11 `uint32_t audio_buffer_t::buffer_error`

23.1.6.0.0.52.12 `semaphore_t audio_buffer_t::sem`

23.1.7 struct sound_card_t

Data Fields

- `audio_controller_t controller`
Controller.
- `audio_codec_t codec`
Codec.
- `audio_buffer_t buffer`
Audio buffer managed by the Soundcard.

23.1.7.0.0.53 Field Documentation

23.1.7.0.0.53.1 `audio_buffer_t sound_card_t::buffer`

23.2 Macro Definition Documentation

23.2.1 #define USEDMA 1

23.3 Enumeration Type Documentation

23.3.1 enum snd_status_t

Enumerator

- kStatus_SND_Success* Execute successfully.
- kStatus_SND_Fail* Execute fail.
- kStatus_SND_DmaFail* DMA operation fail.
- kStatus_SND_CtrlFail* Audio controller operation fail.
- kStatus_SND_CodecFail* Audio codec operation fail.
- kStatus_SND_BufferAllocateFail* Buffer allocate failure.

23.4 Function Documentation

23.4.1 `snd_status_t SND_TxInit (sound_card_t * card, void * ctrl_config, void * codec_config, ctrl_state_t * state)`

The function initializes the controller and the codec.

The function initializes the generic layer structure, for example the buffer and the status structure. Then, the function calls the initialize functions of the controller and the codec.

Parameters

<i>card</i>	Soundcard pointer
<i>ctrl_config</i>	The configuration structure of the audio controller
<i>codec_config</i>	The configuration structure of the audio Codec
<i>state</i>	The audio controller state structure needed in transfer.

Returns

Return *kStatus_SND_Success* while the initialize success and *kStatus_SND_fail* if failed.

23.4.2 `snd_status_t SND_RxInit (sound_card_t * card, void * ctrl_config, void * codec_config, ctrl_state_t * state)`

The function initializes the controller and the codec.

The function initializes the generic layer structure, for example the buffer and the status structure. Then, the function calls the initialize functions of the controller and the codec.

Function Documentation

Parameters

<i>card</i>	Soundcard pointer
<i>ctrl_config</i>	The configuration structure of the audio controller
<i>codec_config</i>	The configuration structure of the audio Codec
<i>state</i>	The audio controller state structure needed in transfer.

Returns

Return kStatus_SND_Success while the initialize success and kStatus_SND_fail if failed.

23.4.3 `snd_status_t SND_TxDeinit(sound_card_t * card)`

The function calls the codec and controller deinitialization function and frees the buffer controlled by the Soundcard. The function should be used at the end of the application. If the playback/record is paused, do not use the function. Instead, use the snd_stop_tx/snd_stop_rx.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

Returns

Return kStatus_SND_Success while the initialize success and kStatus_SND_fail if failed.

23.4.4 `snd_status_t SND_RxDeinit(sound_card_t * card)`

The function calls the codec and the controller deinitialization function and frees the buffer controlled by the Soundcard. The function should be used at the end of the application. If the playback/record is paused, do not use the function. Instead, use the snd_stop_tx/snd_stop_rx.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

Returns

Return kStatus_SND_Success while the initialize success and kStatus_SND_fail if failed.

23.4.5 **snd_status_t SND_TxConfigDataFormat (sound_card_t * *card*, ctrl_data_format_t * *format*)**

This function can make the application change the data format during run time. This function cannot be called while either the TCSR.TE or RCSR.RE are enabled. This function can change the sample rate, bit depth, such as the 16-bit.

Parameters

<i>card</i>	Soundcard pointer
<i>format</i>	Data format used in the sound card

Returns

Return kStatus_SND_Success while the initialize success and kStatus_SND_fail if failed.

23.4.6 **snd_status_t SND_RxConfigDataFormat (sound_card_t * *card*, ctrl_data_format_t * *format*)**

This function can make the application change the data format during the run time. This function cannot be called while either the TCSR.TE or the RCSR.RE are enabled. This function can change the sample rate, bit depth(i.e. 16-bit).

Parameters

<i>card</i>	Soundcard pointer
<i>format</i>	Data format used in the sound card

Returns

Return kStatus_SND_Success while the initialize success and kStatus_SND_fail if failed.

23.4.7 **uint32_t SND_TxUpdateStatus (sound_card_t * *card*, uint32_t *len*)**

This function should be called after the application copied data into the buffer provided by the Soundcard. The Soundcard does not copy data to the internal buffer. This operation should be done by the applications. The Soundcard provides an interface ,snd_get_status(), for an application to get the information about the internal buffer status, including the starting address and empty blocks.

Function Documentation

Parameters

<i>card</i>	Soundcard pointer
<i>len</i>	Data size of the data to write

Returns

The size which has been written.

23.4.8 uint32_t SND_RxUpdateStatus (sound_card_t * *card*, uint32_t *len*)

This function should be called after the application copied data into the buffer provided by the Soundcard. The Soundcard does not help users copy data to the internal buffer. This operation should be done by the applications. The Soundcard provides an interface ,snd_get_status(), for an application to get the information about the internal buffer status, including the starting address and empty blocks and so on.

Parameters

<i>card</i>	Soundcard pointer
<i>len</i>	Data size of the data to write

Returns

The size which has been written.

23.4.9 void SND_GetStatus (sound_card_t * *card*, snd_state_t * *status*)

Each time the application wants to write/read data from the internal buffer, it should call this function to get the status of the internal buffer. This function copies data to the

Parameters

<i>status</i>	from the structure. The user can get the information about where to write/read data and how much data can be read/written.
---------------	--

<i>card</i>	Soundcard pointer
<i>status</i>	Pointer of the audio_status_t structure
<i>card</i>	Soundcard pointer

23.4.10 void SND_TxStart (sound_card_t * *card*)

This function starts the Tx process of the Soundcard. This function enables the DMA/interrupt request source and enables the Tx and the bit clock of the Tx. Note that this function can be used both in the beginning of the SAI transfer and also resume the transfer.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

23.4.11 void SND_RxStart (sound_card_t * *card*)

This function starts the Rx process of the Soundcard. This function enables the DMA/interrupt request source and enables the Tx and the bit clock of the Tx. Note that this function can be used both in the beginning of the SAI transfer and also resume the transfer.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

23.4.12 static void SND_TxStop (sound_card_t * *card*) [inline], [static]

This function stops the transfer of the Soundcard Tx. Note that this function does not close the audio controller. It disables the DMA/interrupt request source. Therefore, this function can be used to pause the audio play.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

23.4.13 static void SND_RxStop (sound_card_t * *card*) [inline], [static]

This function stops the transfer of the Soundcard Rx. Note that this function does not close the audio controller. It disables the DMA/interrupt request source. Therefore, this function can be used to pause the

Function Documentation

audio record.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

23.4.14 void SND_WaitEvent (sound_card_t * *card*)

The application should call this function before write/read data from the Soundcard buffer. Before the application writes data to the Soundcard buffer, the buffer must have free space for the new data. Otherwise, data loss occurs. This function waits for the semaphore which represents the free space in the Soundcard buffer. Similarly to the reading data from the Soundcard buffer, effective data must be in the buffer. This function waits for that semaphore.

Parameters

<i>card</i>	Soundcard pointer
-------------	-------------------

23.4.15 snd_status_t SND_SetMuteCmd (sound_card_t * *card*, bool *enable*)

This interface sets the mute option for the Soundcard.

Parameters

<i>card</i>	Soundcard pointer
<i>enable</i>	Mute or unmute. True means mute, false means unmute.

23.4.16 snd_status_t SND_SetVolume (sound_card_t * *card*, uint32_t *volume*)

This interface sets the volume of Soundcard.

Parameters

<i>card</i>	Soundcard pointer.
<i>volume</i>	Volume of the Soundcard.

23.4.17 uint32_t SND_GetVolume (sound_card_t * *card*)

Function Documentation

Parameters

<i>card</i>	Soundcard pointer.
-------------	--------------------

Returns

Voulme number of Soundcard.

Chapter 24

Serial Peripheral Interface (SPI)

The Kinetis SDK provides both HAL and Peripheral drivers for the Serial Peripheral Interface (SPI) block of Kinetis devices.

Modules

- [SPI Classes](#)
This part describes the SPI driver C++ classes.
- [SPI HAL driver](#)
This part describes the programming interface of the SPI HAL driver.
- [SPI Master Peripheral Driver](#)
This part describes the programming interface of the SPI master mode Peripheral driver.
- [SPI Slave Peripheral Driver](#)
This part describes the programming interface of the SPI slave mode Peripheral driver.
- [Shared SPI Types](#)
This part describes SPI driver shared types.

24.0.18 SPI Master Peripheral Driver

Overview

The SPI master mode Peripheral driver transfers data to and from the external devices on the SPI bus in master mode. It provides an easy way to transfer buffers of data with a single function call.

Run-time state structures

The SPI master driver uses a run-time state structure, [spi_master_state_t](#), to track the ongoing data transfer. The structure holds data that the SPI master peripheral driver uses to communicate between the transfer function and the interrupt handler. The interrupt handler also uses this information to keep track of its progress. The user is only responsible to pass the memory for this run-time state structure and the SPI master driver fills out the members.

Device structures

The driver uses instances of the `#spi_device_t` structure to represent the SPI bus configuration required to communicate the bus connectivity to an external device.

The device structures can be passed to the data transfer functions and the bus is reconfigured before the transfer is started. Alternatively, you can manually configure the SPI bus for a device. For example, if there is only one device connected to the bus, you might configure it only once.

Initialization

To initialize the SPI master driver, call the [SPI_DRV_MasterInit\(\)](#) function and pass the instance number of the SPI peripheral you want to use. For example, to use the SPI1, pass a value of 1 to the initialization function.

The user passes in a device structure that represents the characteristics of the SPI bus including the desired baud rate. Note that, in some cases, the exact baud rate cannot be achieved. Instead, the closest matching baud rate is returned via the calculatedBaudRate parameter. Note that this parameter never exceeds the desired baud rate. An example device structure usage and SPI initialization is as follows:

This is an example code to initialize and configure the driver:

```
uint32_t masterInstance = 1;
spi_master_state_t spiMasterState;
uint32_t calculatedBaudRate;

// Define bus configuration.
spi_device_t device = {
    .bitsPerSec = 1000,
    .polarity = kSpiClockPolarity_ActiveHigh,
    .phase = kSpiClockPhase_SecondEdge,
    .direction = kSpiMsbFirst
};

// Init driver.
SPI_DRV_MasterInit(masterInstance, &spiMasterState);

// Configure bus.
SPI_DRV_MasterConfigureBus(masterInstance, &device, &calculatedBaudRate);
```

Additionally, the SPI supports DMA transfers. To use the SPI with DMA, call an alternate initialization function, [SPI_DRV_MasterInitDma](#). The user still needs to call the [SPI_DRV_MasterConfigureBus\(\)](#) function. In addition to initialization of the SPI interface, it is also necessary to configure the SPI to match the parameters needed by the peripheral device as mentioned above.

```
/* Function prototype */
void SPI_DRV_MasterInitDma(uint32_t instance,
                           spi_master_state_t * spiState);

SPI_DRV_MasterConfigureBus(masterInstance, &device, &calculatedBaudRate);
```

Transfers

The driver supports two different modes to transfer data: blocking and non-blocking. The blocking transfer function is the [SPI_DRV_MasterTransfer\(\)](#).

This is an example of a blocking transfer:

```
uint8_t sendBuf[] = { 0, 1, 2, 3 };
uint8_t receiveBuf[sizeof(sendBuf)];

SPI_DRV_MasterTransfer(masterInstance, // SPI peripheral instance
                      &device,           // Device SPI bus configuration
                      sendBuf,          // Data to send
                      receiveBuf,        // Buffer to hold received data
                      sizeof(sendBuf),   // Number of bytes to transfer
                      kSpiWaitForever); // No timeout
```

This is an example of a non-blocking transfer:

```
uint8_t sendBuf[] = { 0, 1, 2, 3 };
uint8_t receiveBuf[sizeof(sendBuf)];

// Start the transfer.
SPI_DRV_MasterTransfer_async(masterInstance, // SPI peripheral instance
    &device, // Device SPI bus configuration
    sendBuf, // Data to send
    receiveBuf, // Buffer to hold received data
    sizeof(sendBuf)); // Number of bytes to transfer

// Wait for the transfer to complete.
while (SPI_DRV_MasterGetTransferStatus(0, NULL) ==
    kStatus_SPI_Busy)
{
}
```

For non-blocking/async transfers, check back to get the transfer status as shown above. Another example where framesXfer returns the number of frames transferred:

```
SPI_DRV_MasterGetTransferStatus(masterInstance, &framesXfer);
```

To abort a transfer, simply call:

```
spi_status_t SPI_DRV_MasterAbortTransfer(masterInstance);
```

De-initialization

To de-initialize or shut down the SPI module, call the function:

```
void SPI_DRV_MasterDeinit(masterInstance);
```

SPI HAL driver

24.1 SPI HAL driver

This chapter describes the programming interface of the SPI HAL driver.

Enumerations

- enum `spi_status_t` {
 `kStatus_SPI_SlaveTxUnderrun`,
 `kStatus_SPI_SlaveRxOverrun`,
 `kStatus_SPI_Timeout`,
 `kStatus_SPI_Busy`,
 `kStatus_SPI_NoTransferInProgress` }
Error codes for the SPI driver.
- enum `spi_master_slave_mode_t` {
 `kSpiMaster` = 1,
 `kSpiSlave` = 0 }
SPI master or slave configuration.
- enum `spi_clock_polarity_t` {
 `kSpiClockPolarity_ActiveHigh` = 0,
 `kSpiClockPolarity_ActiveLow` = 1 }
SPI clock polarity configuration.
- enum `spi_clock_phase_t` {
 `kSpiClockPhase_FirstEdge` = 0,
 `kSpiClockPhase_SecondEdge` = 1 }
SPI clock phase configuration.
- enum `spi_shift_direction_t` {
 `kSpiMsbFirst` = 0,
 `kSpiLsbFirst` = 1 }
SPI data shifter direction options.
- enum `spi_ss_output_mode_t` {
 `kSpiSlaveSelect_AsGpio` = 0,
 `kSpiSlaveSelect_FaultInput` = 2,
 `kSpiSlaveSelect_AutomaticOutput` = 3 }
SPI slave select output mode options.
- enum `spi_pin_mode_t` {
 `kSpiPinMode_Normal` = 0,
 `kSpiPinMode_Input` = 1,
 `kSpiPinMode_Output` = 3 }
SPI pin mode options.

Configuration

- void `SPI_HAL_Init` (uint32_t baseAddr)
Restores the SPI to reset configuration.
- static void `SPI_HAL_Enable` (uint32_t baseAddr)
Enables the SPI peripheral.

- static void **SPI_HAL_Disable** (uint32_t baseAddr)
Disables the SPI peripheral.
- uint32_t **SPI_HAL_SetBaud** (uint32_t baseAddr, uint32_t bitsPerSec, uint32_t sourceClockInHz)
Sets the SPI baud rate in bits per second.
- static void **SPI_HAL_SetBaudDivisors** (uint32_t baseAddr, uint32_t prescaleDivisor, uint32_t rateDivisor)
Configures the baud rate divisors manually.
- static void **SPI_HAL_SetMasterSlave** (uint32_t baseAddr, **spi_master_slave_mode_t** mode)
Configures the SPI for master or slave.
- static bool **SPI_HAL_IsMaster** (uint32_t baseAddr)
Returns whether the SPI module is in master mode.
- void **SPI_HAL_SetSlaveSelectOutputMode** (uint32_t baseAddr, **spi_ss_output_mode_t** mode)
Sets how the slave select output operates.
- void **SPI_HAL_SetDataFormat** (uint32_t baseAddr, **spi_clock_polarity_t** polarity, **spi_clock_phase_t** phase, **spi_shift_direction_t** direction)
Sets the polarity, phase, and shift direction.
- void **SPI_HAL_SetPinMode** (uint32_t baseAddr, **spi_pin_mode_t** mode)
Sets the SPI pin mode.

DMA

- void **SPI_HAL_SetTxDmaCmd** (uint32_t baseAddr, bool enableTransmit)
Configures the transmit DMA request.
- void **SPI_HAL_SetRxDmaCmd** (uint32_t baseAddr, bool enableReceive)
Configures the receive DMA requests.

Low power

- static void **SPI_HAL_ConfigureStopInWaitMode** (uint32_t baseAddr, bool enable)
Enables or disables the SPI clock to stop when the CPU enters wait mode.

Interrupts

- static void **SPI_HAL_SetReceiveAndFaultIntCmd** (uint32_t baseAddr, bool enable)
Enables or disables the SPI receive buffer full and mode fault interrupt.
- static void **SPI_HAL_SetTransmitIntCmd** (uint32_t baseAddr, bool enable)
Enables or disables the SPI transmit buffer empty interrupt.
- static void **SPI_HAL_SetMatchIntCmd** (uint32_t baseAddr, bool enable)
Enables or disables the SPI match interrupt.

Status

- static bool **SPI_HAL_IsReadBuffFullPending** (uint32_t baseAddr)
Checks whether the read buffer is full.
- static bool **SPI_HAL_IsTxBuffEmptyPending** (uint32_t baseAddr)

SPI HAL driver

- static bool **SPI_HAL_IsModeFaultPending** (uint32_t baseAddr)
Checks whether the transmit buffer is empty.
- void **SPI_HAL_ClearModeFaultFlag** (uint32_t baseAddr)
Checks whether a mode fault occurred.
- static bool **SPI_HAL_IsMatchPending** (uint32_t baseAddr)
Clears the mode fault flag.
- void **SPI_HAL_ClearMatchFlag** (uint32_t baseAddr)
Checks whether the data received matches the previously-set match value.
- void **SPI_HAL_ClearMatchFlag** (uint32_t baseAddr)
Clears the match flag.

Data transfer

- static uint8_t **SPI_HAL_ReadData** (uint32_t baseAddr)
Reads a byte from the data buffer.
- static void **SPI_HAL_WriteData** (uint32_t baseAddr, uint8_t data)
Writes a byte into the data buffer.
- void **SPI_HAL_WriteDataBlocking** (uint32_t baseAddr, uint8_t data)
Writes a byte into the data buffer and waits till complete to return.

Match byte

- static void **SPI_HAL_SetMatchValue** (uint32_t baseAddr, uint8_t matchByte)
Sets the value which triggers the match interrupt.

24.1.1 Enumeration Type Documentation

24.1.1.1 enum spi_status_t

Enumerator

kStatus_SPI_SlaveTxUnderrun SPI Slave TX Underrun error.

kStatus_SPI_SlaveRxOverrun SPI Slave RX Overrun error.

kStatus_SPI_Timeout SPI transfer timed out.

kStatus_SPI_Busy SPI instance is already busy performing a transfer.

kStatus_SPI_NoTransferInProgress Attempt to abort a transfer when no transfer was in progress.

24.1.1.2 enum spi_master_slave_mode_t

Enumerator

kSpiMaster SPI peripheral operates in master mode.

kSpiSlave SPI peripheral operates in slave mode.

24.1.1.3 enum spi_clock_polarity_t

Enumerator

kSpiClockPolarity_ActiveHigh Active-high SPI clock (idles low).

kSpiClockPolarity_ActiveLow Active-low SPI clock (idles high).

24.1.1.4 enum spi_clock_phase_t

Enumerator

kSpiClockPhase_FirstEdge First edge on SPSCK occurs at the middle of the first cycle of a data transfer.

kSpiClockPhase_SecondEdge First edge on SPSCK occurs at the start of the first cycle of a data transfer.

24.1.1.5 enum spi_shift_direction_t

Enumerator

kSpiMsbFirst Data transfers start with most significant bit.

kSpiLsbFirst Data transfers start with least significant bit.

24.1.1.6 enum spi_ss_output_mode_t

Enumerator

kSpiSlaveSelect_AsGpio Slave select pin configured as GPIO.

kSpiSlaveSelect_FaultInput Slave select pin configured for fault detection.

kSpiSlaveSelect_AutomaticOutput Slave select pin configured for automatic SPI output.

24.1.1.7 enum spi_pin_mode_t

Enumerator

kSpiPinMode_Normal Pins operate in normal, single-direction mode.

kSpiPinMode_Input Bidirectional mode. Master: MOSI pin is input; Slave: MISO pin is input

kSpiPinMode_Output Bidirectional mode. Master: MOSI pin is output; Slave: MISO pin is output

24.1.2 Function Documentation

24.1.2.1 void SPI_HAL_Init (uint32_t baseAddr)

This function basically resets all of the SPI registers to their default setting including disabling the module.

SPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

24.1.2.2 static void SPI_HAL_Enable (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

24.1.2.3 static void SPI_HAL_Disable (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

24.1.2.4 uint32_t SPI_HAL_SetBaud (uint32_t *baseAddr*, uint32_t *bitsPerSec*, uint32_t *sourceClockInHz*)

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

Parameters

<i>baseAddr</i>	Module base address
<i>bitsPerSec</i>	The desired baud rate in bits per second
<i>sourceClockInHz</i>	Module source input clock in Hertz

Returns

The actual calculated baud rate

24.1.2.5 static void SPI_HAL_SetBaudDivisors (uint32_t *baseAddr*, uint32_t *prescaleDivisor*, uint32_t *rateDivisor*) [inline], [static]

This function allows the caller to manually set the baud rate divisors in the event that these dividers are known and the caller does not wish to call the SPI_HAL_SetBaudRate function.

Parameters

<i>baseAddr</i>	Module base address
<i>prescale-Divisor</i>	baud rate prescale divisor setting
<i>rateDivisor</i>	baud rate divisor setting

24.1.2.6 static void SPI_HAL_SetMasterSlave (uint32_t *baseAddr*, spi_master_slave_mode_t *mode*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>mode</i>	Mode setting (master or slave) of type dspi_master_slave_mode_t

24.1.2.7 static bool SPI_HAL_IsMaster (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

Return values

<i>true</i>	The module is in master mode.
<i>false</i>	The module is in slave mode.

24.1.2.8 void SPI_HAL_SetSlaveSelectOutputMode (uint32_t *baseAddr*, spi_ss_output_mode_t *mode*)

This function allows the user to configure the slave select in one of the three operational modes: as GPIO, as a fault input, or as an automatic output for standard SPI modes.

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

SPI HAL driver

<i>mode</i>	Selection input of one of three modes of type spi_ss_output_mode_t
-------------	--

24.1.2.9 void SPI_HAL_SetDataFormat (uint32_t *baseAddr*, spi_clock_polarity_t *polarity*, spi_clock_phase_t *phase*, spi_shift_direction_t *direction*)

This function configures the clock polarity, clock phase, and data shift direction.

Parameters

<i>baseAddr</i>	Module base address
<i>polarity</i>	Clock polarity setting of type spi_clock_polarity_t.
<i>phase</i>	Clock phase setting of type spi_clock_phase_t.
<i>direction</i>	Data shift direction (MSB or LSB) of type spi_shift_direction_t.

24.1.2.10 void SPI_HAL_SetPinMode (uint32_t *baseAddr*, spi_pin_mode_t *mode*)

This function configures the SPI data pins to one of three modes (of type spi_pin_mode_t): Single direction mode: MOSI and MISO pins operate in normal, single direction mode. Bidirectional mode: Master: MOSI configured as input, Slave: MISO configured as input. Bidirectional mode: Master: MOSI configured as output, Slave: MISO configured as output.

Parameters

<i>baseAddr</i>	Module base address
<i>mode</i>	Operational of SPI pins of type spi_pin_mode_t.

24.1.2.11 void SPI_HAL_SetTxDmaCmd (uint32_t *baseAddr*, bool *enableTransmit*)

This function enables or disables the SPI TX DMA request. When the TX DMA is enabled it disables the TX interrupt.

Parameters

<i>baseAddr</i>	Module base address
<i>enableTransmit</i>	Enable (true) or disable (false) the transmit DMA request.

24.1.2.12 void SPI_HAL_SetRxDmaCmd (uint32_t *baseAddr*, bool *enableReceive*)

This function enables or disables the SPI RX DMA request. When the RX DMA is enabled it disables the RX interrupt.

SPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
<i>enableReceive</i>	Enable (true) or disable (false) the receive DMA request.

24.1.2.13 static void SPI_HAL_ConfigureStopInWaitMode (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables or disables the SPI clock operation in wait mode.

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enable (true) or disable (false) the SPI clock in wait mode.

24.1.2.14 static void SPI_HAL_SetReceiveAndFaultIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables or disables the SPI receive buffer full and mode fault interrupt.

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enable (true) or disable (false) the receive buffer full and mode fault interrupt.

24.1.2.15 static void SPI_HAL_SetTransmitIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables or disables the SPI transmit buffer empty interrupt.

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enable (true) or disable (false) the transmit buffer empty interrupt.

24.1.2.16 static void SPI_HAL_SetMatchIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables or disables the SPI match interrupt.

Parameters

<i>baseAddr</i>	Module base address
<i>enable</i>	Enable (true) or disable (false) the match interrupt.

24.1.2.17 static bool SPI_HAL_IsReadBuffFullPending (uint32_t *baseAddr*) [inline], [static]

The read buffer full flag is only cleared by reading it when it is set, then reading the data register by calling the [SPI_HAL_ReadData\(\)](#). This example code demonstrates how to check the flag, read data, and clear the flag.

```
// Check read buffer flag.
if (SPI_HAL_IsReadBuffFullPending(0))
{
    // Read the data in the buffer, which also clears the flag.
    byte = SPI_HAL_ReadData(0);
}
```

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

Return values

<i>Current</i>	setting of the read buffer full flag
----------------	--------------------------------------

24.1.2.18 static bool SPI_HAL_IsTxBuffEmptyPending (uint32_t *baseAddr*) [inline], [static]

To clear the transmit buffer empty flag, you must first read the flag when it is set. Then write a new data value into the transmit buffer with a call to the [SPI_HAL_WriteData\(\)](#). The example code shows how to do this.

```
// Check if transmit buffer is empty.
if (SPI_HAL_IsTxBuffEmptyPending(0))
{
    // Buffer has room, so write the next data value.
    SPI_HAL_WriteData(0, byte);
}
```

SPI HAL driver

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

Return values

<i>Current</i>	setting of the transmit buffer empty flag
----------------	---

24.1.2.19 static bool SPI_HAL_IsModeFaultPending (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

Return values

<i>Current</i>	setting of the mode fault flag
----------------	--------------------------------

24.1.2.20 void SPI_HAL_ClearModeFaultFlag (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

24.1.2.21 static bool SPI_HAL_IsMatchPending (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

Return values

<i>Current</i>	setting of the match flag
----------------	---------------------------

24.1.2.22 void SPI_HAL_ClearMatchFlag (uint32_t *baseAddr*)

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

24.1.2.23 static uint8_t SPI_HAL_ReadData (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
-----------------	---------------------

24.1.2.24 static void SPI_HAL_WriteData (uint32_t *baseAddr*, uint8_t *data*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>data</i>	The data to send

24.1.2.25 void SPI_HAL_WriteDataBlocking (uint32_t *baseAddr*, uint8_t *data*)

This function writes data to the data register and waits until the TX is empty to return.

Parameters

<i>baseAddr</i>	Module base address
<i>data</i>	The data to send

24.1.2.26 static void SPI_HAL_SetMatchValue (uint32_t *baseAddr*, uint8_t *matchByte*) [inline], [static]

Parameters

<i>baseAddr</i>	Module base address
<i>matchByte</i>	The value which triggers the match interrupt.

SPI Master Peripheral Driver

24.2 SPI Master Peripheral Driver

This chapter describes the programming interface of the SPI master mode Peripheral driver.

Data Structures

- struct [spi_master_user_config_t](#)
Information about a device on the SPI bus. [More...](#)
- struct [spi_master_state_t](#)
Runtime state of the SPI master driver. [More...](#)

Enumerations

- enum [_spi_timeouts](#) { [kSpiWaitForever](#) = 0xffffffffU }

Initialization and shutdown

- void [SPI_DRV_MasterInit](#) (uint32_t instance, [spi_master_state_t](#) *spiState)
Initializes an SPI instance for master mode operation.
- void [SPI_DRV_MasterInitDma](#) (uint32_t instance, [spi_master_state_t](#) *spiState)
Initializes a SPI instance for master mode operation with DMA support.
- void [SPI_DRV_MasterDeinit](#) (uint32_t instance)
Shuts down a SPI instance.

Bus configuration

- void [SPI_DRV_MasterConfigureBus](#) (uint32_t instance, const [spi_master_user_config_t](#) *device, uint32_t *calculatedBaudRate)
Configures the SPI port to access a device on the bus.

Blocking transfers

- [spi_status_t SPI_DRV_MasterTransferBlocking](#) (uint32_t instance, const [spi_master_user_config_t](#) *restrict device, const uint8_t *restrict sendBuffer, uint8_t *restrict receiveBuffer, size_t transferByteCount, uint32_t timeout)
Performs a blocking SPI master mode transfer.

Non-blocking transfers

- [spi_status_t SPI_DRV_MasterTransfer](#) (uint32_t instance, const [spi_master_user_config_t](#) *restrict device, const uint8_t *restrict sendBuffer, uint8_t *restrict receiveBuffer, size_t transferByteCount)
Performs a non-blocking SPI master mode transfer.
- [spi_status_t SPI_DRV_MasterGetTransferStatus](#) (uint32_t instance, uint32_t *bytesTransferred)

- **spi_status_t SPI_DRV_MasterAbortTransfer (uint32_t instance)**
Returns whether the previous transfer finished.
Terminates an asynchronous transfer early.

24.2.1 Data Structure Documentation

24.2.1.1 struct spi_master_user_config_t

Data Fields

- **uint32_t bitsPerSec**
SPI baud rate in bits per sec.

24.2.1.2 struct spi_master_state_t

This structure holds data that are used by the SPI master peripheral driver to communicate between the transfer function and the interrupt handler. The interrupt handler also uses this information to keep track of its progress.

Data Fields

- **uint32_t spiSourceClock**
Module source clock.
- **volatile bool isTransferInProgress**
True if there is an active transfer.
- **bool isTransferAsync**
Whether the transfer is asynchronous.
- **const uint8_t *restrict sendBuffer**
The buffer being sent.
- **uint8_t *restrict receiveBuffer**
The buffer into which received bytes are placed.
- **volatile size_t remainingSendByteCount**
Number of bytes remaining to send.
- **volatile size_t remainingReceiveByteCount**
Number of bytes remaining to receive.
- **volatile size_t transferredByteCount**
Number of bytes transferred so far.
- **semaphore_t irqSync**
Used to wait for ISR to complete its business.

SPI Master Peripheral Driver

24.2.1.2.0.54 Field Documentation

24.2.1.2.0.54.1 `volatile bool spi_master_state_t::isTransferInProgress`

24.2.1.2.0.54.2 `bool spi_master_state_t::isTransferAsync`

24.2.1.2.0.54.3 `const uint8_t* restrict spi_master_state_t::sendBuffer`

24.2.1.2.0.54.4 `uint8_t* restrict spi_master_state_t::receiveBuffer`

24.2.1.2.0.54.5 `volatile size_t spi_master_state_t::remainingSendByteCount`

24.2.1.2.0.54.6 `volatile size_t spi_master_state_t::remainingReceiveByteCount`

24.2.1.2.0.54.7 `volatile size_t spi_master_state_t::transferredByteCount`

24.2.1.2.0.54.8 `semaphore_t spi_master_state_t::irqSync`

24.2.2 Enumeration Type Documentation

24.2.2.1 enum _spi_timeouts

Enumerator

`kSpiWaitForever` Waits forever for a transfer to complete.

24.2.3 Function Documentation

24.2.3.1 `void SPI_DRV_MasterInit(uint32_t instance, spi_master_state_t * spiState)`

This function uses a CPU interrupt driven method for transferring data. This function initializes the run-time state structure to track the ongoing transfers, ungates the clock to the SPI module, resets and initializes the module to default settings, configures the IRQ state structure, enables the module-level interrupt to the core, and enables the SPI module.

Parameters

<code>instance</code>	The instance number of the SPI peripheral.
<code>spiState</code>	The pointer to the SPI master driver state structure. The user must pass the memory for this run-time state structure and the SPI master driver fills out the members. This run-time state structure keeps track of the transfer in progress.

24.2.3.2 void SPI_DRV_MasterInitDma (uint32_t *instance*, spi_master_state_t * *spiState*)

This function is exactly like the `spi_master_init` function except that it also adds the DMA support. To use the DMA-based transfers, use this function call instead of the `spi_master_init` function call. Like the `spi_master_init`, this function initializes the run-time state structure to track the ongoing transfers, ungates the clock to the SPI module, resets the SPI module, initializes the module to user defined settings and default settings, configures the IRQ state structure, enables the module-level interrupt to the core, and enables the SPI module.

This initialization function also configures the DMA module by requesting channels for DMA operation and sets a "useDma" flag in the run-time state structure to notify transfer functions to use DMA driven operations.

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
<i>spiState</i>	The pointer to the SPI master driver state structure. The user must pass the memory for this run-time state structure and the SPI master driver fills out the members. This run-time state structure keeps track of the transfer in progress.

24.2.3.3 void SPI_DRV_MasterDeinit (uint32_t *instance*)

This function resets the SPI peripheral, gates its clock, and disables the interrupt to the core.

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
-----------------	--

24.2.3.4 void SPI_DRV_MasterConfigureBus (uint32_t *instance*, const spi_master_user_config_t * *device*, uint32_t * *calculatedBaudRate*)

The term "device" is used to indicate the SPI device for which the SPI master is communicating. The user has two options to configure the device parameters: either pass in the pointer to the device configuration structure to the desired transfer function (see `SPI_DRV_MasterTransferDataBlocking` or `SPI_DRV_MasterTransferData`) or pass it in to the `SPI_DRV_MasterConfigureBus` function. The user can pass in a device structure to the transfer function which contains the parameters for the bus (the transfer function then calls this function). However, the user has the option to call this function directly especially to get the calculated baud rate, at which point they may pass in NULL for the device structure in the transfer function (assuming they have called this configure bus function first).

SPI Master Peripheral Driver

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
<i>device</i>	Pointer to the device information structure. This structure contains the settings for SPI bus configurations.
<i>calculated-BaudRate</i>	The calculated baud rate passed back to the user to determine if the calculated baud rate is close enough to meet the needs. The baud rate never exceeds the desired baud rate.

24.2.3.5 **spi_status_t SPI_DRV_MasterTransferBlocking (uint32_t *instance*, const spi_master_user_config_t *restrict *device*, const uint8_t *restrict *sendBuffer*, uint8_t *restrict *receiveBuffer*, size_t *transferByteCount*, uint32_t *timeout*)**

This function simultaneously sends and receives data on the SPI bus, as SPI is naturally a full-duplex bus. The function does return until the transfer is complete.

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
<i>device</i>	Pointer to the device information structure. This structure contains the settings for the SPI bus configuration for this transfer. You may pass NULL for this parameter, in which case the current bus configuration is used unmodified.
<i>sendBuffer</i>	Buffer of data to send. You may pass NULL for this parameter, in which case bytes with a value of 0 (zero) are sent.
<i>receiveBuffer</i>	Buffer where received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
<i>transferByte-Count</i>	The number of bytes to send and receive.
<i>timeout</i>	A timeout for the transfer in microseconds. If the transfer takes longer than this amount of time, the transfer is aborted and a kStatus_SPI_Timeout error is returned.

Return values

# kStatus_Success	The transfer was successful.
kStatus_SPI_Busy	Cannot perform another transfer because one is already in progress.

<i>kStatus_SPI_Timeout</i>	The transfer timed out and was aborted.
----------------------------	---

24.2.3.6 **spi_status_t SPI_DRV_MasterTransfer (uint32_t *instance*, const spi_master_user_config_t *restrict *device*, const uint8_t *restrict *sendBuffer*, uint8_t *restrict *receiveBuffer*, size_t *transferByteCount*)**

This function returns immediately. It is the user's responsibility to check back to ascertain if the transfer is complete (using the SPI_DRV_MasterGetTransferStatus function). This function simultaneously sends and receives data on the SPI bus, as SPI is naturally a full-duplex bus. The function does return until the transfer is complete.

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
<i>device</i>	Pointer to the device information structure. This structure contains the settings for the SPI bus configuration for this transfer. You may pass NULL for this parameter, in which case the current bus configuration is used unmodified.
<i>sendBuffer</i>	Buffer of data to send. You may pass NULL for this parameter, in which case bytes with a value of 0 (zero) is sent.
<i>receiveBuffer</i>	Buffer where received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
<i>transferByteCount</i>	The number of bytes to send and receive.

Return values

# <i>kStatus_Success</i>	The transfer was successful.
<i>kStatus_SPI_Busy</i>	Cannot perform another transfer because one is already in progress.
<i>kStatus_SPI_Timeout</i>	The transfer timed out and was aborted.

24.2.3.7 **spi_status_t SPI_DRV_MasterGetTransferStatus (uint32_t *instance*, uint32_t * *bytesTransferred*)**

When performing an a-sync transfer, the user can call this function to ascertain the state of the current transfer: in progress (or busy) or complete (success). In addition, if the transfer is still in progress, the user can get the number of words that have been transferred up to now.

SPI Master Peripheral Driver

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
<i>bytes-Transferred</i>	Pointer to a value that is filled in with the number of bytes that were sent in the active transfer

Return values

<i>kStatus_Success</i>	The transfer has completed successfully.
<i>kStatus_SPI_Busy</i>	The transfer is still in progress. <i>bytesTransferred</i> is filled with the number of bytes that have been transferred so far.

24.2.3.8 `spi_status_t SPI_DRV_MasterAbortTransfer(uint32_t instance)`

During an async transfer, the user has the option to terminate the transfer early if the transfer is still in progress.

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
-----------------	--

Return values

<i>kStatus_SPI_Success</i>	The transfer was successful.
<i>kStatus_SPI_NoTransfer-InProgress</i>	No transfer is currently in progress.

24.3 SPI Slave Peripheral Driver

This chapter describes the programming interface of the SPI slave mode Peripheral driver.

Data Structures

- struct [spi_slave_callbacks_t](#)
The set of callbacks used for SPI slave mode. [More...](#)
- struct [spi_slave_state_t](#)
Runtime state of the SPI slave driver. [More...](#)
- struct [spi_slave_user_config_t](#)
Definition of application implemented configuration and callback. [More...](#)

SPI Slave

- void [SPI_DRV_SlaveInit](#) (uint32_t instance, [spi_slave_state_t](#) *spiState, const [spi_slave_user_config_t](#) *config)
Initializes the SPI module for slave mode.
- [spi_status_t SPI_DRV_SlaveInitDma](#) (uint32_t instance, [spi_slave_state_t](#) *spiState, const [spi_slave_user_config_t](#) *slaveConfig, const uint8_t *sendBuffer, uint8_t *receiveBuffer, size_t transferByteCount)
Initializes an SPI instance for slave mode operation with the DMA support.
- void [SPI_DRV_SlaveDeinit](#) (uint32_t instance)
De-initializes the device.

24.3.0.9 SPI Slave Peripheral Driver

Overview

The SPI slave peripheral driver provides an easy way to use an SPI peripheral in slave mode.

Data transfer is performed through callback functions.

Runtime state of the SPI slave driver

This structure of type [spi_slave_state_t](#) holds data that the SPI slave peripheral driver uses to communicate between the transfer function and the interrupt handler. The user needs to pass the memory for this structure and the driver fills out the members.

Callbacks

To use this driver, first define several application callbacks. These are the callback functions that are set in the [spi_slave_callbacks_t](#) structure.

SPI Slave Peripheral Driver

The three callbacks are:

- Data source
- Data sink
- Error notification

The first two callbacks are used to send and receive data. The third callback is invoked if an error occurs.

The prototypes for the three callbacks are:

```
status_t dataSource(uint8_t * sourceByte, uint16_t instance);  
status_t dataSink(uint8_t sinkByte, uint16_t instance);  
void onError(status_t error);
```

All callbacks are invoked from IRQ state.

Setup

SPI Slave Interrupt Driven: To initialize the SPI slave driver, first create and fill in the #spi_slave_config_t structure. This structure defines the callbacks and data format settings for the SPI peripheral. The structure is not required after the driver is initialized and can be allocated on the stack.

This is an example of a config struct definition and SPI slave initialization:

```
spi_slave_user_config_t slaveUserConfig;  
slaveUserConfig.polarity = kSpiClockPolarity_ActiveHigh;  
slaveUserConfig.phase = kSpiClockPhase_FirstEdge;  
slaveUserConfig.direction = kSpiMsbFirst;  
slaveUserConfig.callbacks.dataSink = data_sink;  
slaveUserConfig.callbacks.dataSource = data_source;  
slaveUserConfig.callbacks.onError = on_error;  
  
spi_slave_state_t spiSlaveState;  
  
SPI_DRV_SlaveInit(slaveInstance, &spiSlaveState, &slaveUserConfig);
```

SPI Slave DMA Driven: Additionally, the SPI supports DMA transfers.

The usage of the SPI slave mode with DMA is slightly different than that of the interrupt driven driver. In the case of DMA usage, the user also passes in the send and receive buffer pointers along with the expected amount of data that wish to transfer. This is needed by the DMA engine.

There is no need to define callbacks for the data source and sink. However, an optional callback may be needed to handle errors. Also, an additional new callback is needed from the user to handle the completion of the DMA transfer, which can be as simple as setting a global flag to indicate that the transfer is done. This is an example usage.

To use the SPI with DMA, simply call an alternate initialization function: SPI_DRV_SlaveInitDma:

```
/* Function prototype */  
spi_status_t SPI_DRV_SlaveInitDma(uint32_t instance,
```

```
spi_slave_state_t * spiState,
const spi_slave_user_config_t * slaveConfig,
const uint8_t * sendBuffer,
uint8_t * receiveBuffer,
size_t transferByteCount)
```

Example usage:

```
instance = slaveInstance; <- the desired module instance number
spi_slave_state_t spiSlaveState; <- the user simply allocates memory for this struct
spi_slave_user_config_t slaveUserConfig;
slaveUserConfig.callbacks.onError = on_error; <- set to user implementation of function
slaveUserConfig.callbacks.spi_slave_done_t = spi_Dma_Done; <- user defined callback
slaveUserConfig.polarity = kSpiClockPolarity_ActiveHigh;
slaveUserConfig.phase = kSpiClockPhase_FirstEdge;
slaveUserConfig.direction = kSpiMsbFirst;
sendBuffer <- (pointer) to the source data buffer, can be NULL
receiveBuffer <- (pointer) to the receive data buffer, can be NULL
transferCount <- number of bytes to transfer

SPI_DRV_SlaveInitDma(slaveInstance, &slaveUserConfig, &spiSlaveState,
                     &sendBuffer, &receiveBuffer, transferCount);

/* Here is an example of the callback to indicate completion of the DMA. Note
 * the name of this callback function matches the name used in the structure member above.
 */
uint32_t g_transferDoneFlag = 0;

static void spi_Dma_Done(void)
{
    /* set the global flag to indicate transfer done */
    g_transferDoneFlag = 1;
}
```

De-initialization

To de-initialize or shut down the SPI module, call the function:

```
void SPI_DRV_SlaveDeinit(slaveInstance);
```

24.3.1 Data Structure Documentation

24.3.1.1 struct spi_slave_callbacks_t

Data Fields

- **spi_status_t**(* **dataSource**)(uint8_t *sourceByte, uint32_t instance)
Callback used to get byte to transmit.
- **spi_status_t**(* **dataSink**)(uint8_t sinkByte, uint32_t instance)
Callback used to put received byte.
- void(* **onError**)(spi_status_t error, uint32_t instance)
Callback used to report an SPI error.
- void(* **spi_slave_done_t**)(void)
Callback to report the slave SPI DMA is done transferring data.

SPI Slave Peripheral Driver

24.3.1.1.0.55 Field Documentation

24.3.1.1.0.55.1 `spi_status_t(* spi_slave_callbacks_t::dataSource)(uint8_t *sourceByte, uint32_t instance)`

24.3.1.1.0.55.2 `spi_status_t(* spi_slave_callbacks_t::dataSink)(uint8_t sinkByte, uint32_t instance)`

24.3.1.1.0.55.3 `void(* spi_slave_callbacks_t::onError)(spi_status_t error, uint32_t instance)`

24.3.1.1.0.55.4 `void(* spi_slave_callbacks_t::spi_slave_done_t)(void)`

Used only for DMA enabled slave SPI operation and not used for interrupt operation.

24.3.1.2 struct spi_slave_state_t

This structure holds data that is used by the SPI slave peripheral driver to communicate between the transfer function and the interrupt handler. The user needs to pass in the memory for this structure and the driver fills out the members.

Data Fields

- `uint32_t instance`
DSPI module instance number.
- `spi_slave_callbacks_t callbacks`
Application/user callbacks.

24.3.1.3 struct spi_slave_user_config_t

functions used by the SPI slave driver.

Data Fields

- `spi_slave_callbacks_t callbacks`
Application callbacks.
- `spi_clock_phase_t phase`
Clock phase setting.
- `spi_clock_polarity_t polarity`
Clock polarity setting.
- `spi_shift_direction_t direction`
Either LSB or MSB first.

24.3.1.3.0.56 Field Documentation

24.3.1.3.0.56.1 `spi_slave_callbacks_t spi_slave_user_config_t::callbacks`

24.3.1.3.0.56.2 `spi_clock_phase_t spi_slave_user_config_t::phase`

24.3.1.3.0.56.3 `spi_clock_polarity_t spi_slave_user_config_t::polarity`

24.3.1.3.0.56.4 `spi_shift_direction_t spi_slave_user_config_t::direction`

24.3.2 Function Documentation

24.3.2.1 `void SPI_DRV_SlaveInit(uint32_t instance, spi_slave_state_t * spiState, const spi_slave_user_config_t * config)`

Saves the application callback info, turns on the clock to the module, enables the device, and enables interrupts. Sets the SPI to a slave mode.

SPI Slave Peripheral Driver

Parameters

<i>instance</i>	Instance number of the SPI module.
<i>spiState</i>	The pointer to the SPI slave driver state structure.
<i>config</i>	Pointer to slave mode configuration.

24.3.2.2 `spi_status_t SPI_DRV_SlaveInitDma (uint32_t instance, spi_slave_state_t * spiState, const spi_slave_user_config_t * slaveConfig, const uint8_t * sendBuffer, uint8_t * receiveBuffer, size_t transferByteCount)`

This function is exactly like the SPI_DRV_SlaveInit function except that it also adds the DMA support. This function saves the callbacks to the run-time state structure for later use in the interrupt handler. However, unlike the CPU driven slave driver, there is no need to define callbacks for the data sink or data source since the user passes in buffers for the send and receive data, and the DMA engine uses those buffers. An onError callback is needed to service potential errors seen during a Tx FIFO underflow or Rx FIFO overflow. The user also passes in a user defined callback for handling end of transfers of type spi_slave_done_t. These callbacks are set in the [spi_slave_callbacks_t](#) structure which is part of the [spi_slave_user_config_t](#) structure. See example below. This function also ungates the clock to the SPI module, initializes the SPI for slave mode, enables the module and corresponding interrupts, and sets up the DMA channels. Once initialized, the SPI module is configured in slave mode and ready to receive data from the SPI master. This is an example to set up the [spi_slave_state_t](#) and the [spi_slave_user_config_t](#) parameters and to call the SPI_DRV_SlaveInit function by passing in these parameters: instance = slaveInstance; <- the desired module instance number [spi_slave_state_t](#) spiSlaveState; <- the user simply allocates memory for this struct [spi_slave_user_config_t](#) slaveUserConfig; slaveUserConfig.callbacks.onError = on_error; <- set to user implementation of function slaveUserConfig.callbacks.spi_slave_done_t = spiDmaDone; <- user defined callback slaveUserConfig.dataConfig.clkPhase = kSpiClockPhase_First-Edge; <- example setting slaveUserConfig.dataConfig.clkPolarity = kSpiClockPolarity_ActiveHigh; <- example setting sendBuffer <- (pointer) to the source data buffer, can be NULL receiveBuffer <- (pointer) to the receive data buffer, can be NULL transferCount <- number of bytes to transfer

```
SPI_DRV_SlaveInitDma(slaveInstance, &slaveUserConfig, &spiSlaveState, &sendBuffer, &receiveBuffer, transferCount);
```

Parameters

<i>instance</i>	The instance number of the SPI peripheral.
<i>spiState</i>	The pointer to the SPI slave driver state structure.
<i>slaveConfig</i>	The configuration structure dsPIC33F_Spi.h <code>spi_slave_user_config_t</code> , including the callbacks.

<i>sendBuffer</i>	The pointer to the data buffer of the data to send. You may pass NULL for this parameter, in which case bytes with a value of 0 (zero) are sent.
<i>receiveBuffer</i>	Pointer to the buffer where the received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
<i>transferByte-Count</i>	The expected number of bytes to transfer.

Returns

An error code or kStatus_SPI_Success.

24.3.2.3 void SPI_DRV_SlaveDeinit (uint32_t *instance*)

Clears the control register and turns off the clock to the module.

Parameters

<i>instance</i>	Instance number of the SPI module.
-----------------	------------------------------------

Shared SPI Types

24.4 Shared SPI Types

This chapter describes SPI driver shared types.

24.5 SPI Classes

This chapter describes the SPI driver C++ classes.

Chapter 25

Universal Asynchronous Receiver/Transmitter (UART)

The Kinetis SDK provides both HAL and Peripheral drivers for the Universal Asynchronous Receiver-/Transmitter (UART) block of Kinetis devices.

Modules

- [UART HAL driver](#)

This part describes the programming interface of the UART HAL driver.

- [UART Peripheral Driver](#)

This part describes the programming interface of the UART Peripheral driver.

25.1 UART HAL driver

This chapter describes the programming interface of the UART HAL driver.

Enumerations

- enum `uart_status_t`
Error codes for the UART driver.
- enum `uart_stop_bit_count_t` {
 kUartOneStopBit = 0U,
 kUartTwoStopBit = 1U }
UART number of stop bits.
- enum `uart_parity_mode_t` {
 kUartParityDisabled = 0x0U,
 kUartParityEven = 0x2U,
 kUartParityOdd = 0x3U }
UART parity mode.
- enum `uart_bit_count_per_char_t` {
 kUart8BitsPerChar = 0U,
 kUart9BitsPerChar = 1U }
UART number of bits in a character.
- enum `uart_operation_config_t` {
 kUartOperates = 0U,
 kUartStops = 1U }
UART operation configuration constants.
- enum `uart_receiver_source_t` {
 kUartLoopBack = 0U,
 kUartSingleWire = 1U }
UART receiver source select mode.
- enum `uart_wakeup_method_t` {
 kUartIdleLineWake = 0U,
 kUartAddrMarkWake = 1U }
UART wakeup from standby method constants.
- enum `uart_idle_line_select_t` {
 kUartIdleLineAfterStartBit = 0U,
 kUartIdleLineAfterStopBit = 1U }
UART idle-line detect selection types.
- enum `uart_break_char_length_t` {
 kUartBreakChar10BitMinimum = 0U,
 kUartBreakChar13BitMinimum = 1U }
UART break character length settings for transmit/detect.
- enum `uart_singlewire_txdir_t` {
 kUartSinglewireTxdirIn = 0U,
 kUartSinglewireTxdirOut = 1U }
UART single-wire mode transmit direction.
- enum `uart_ir_tx_pulsewidth_t` {

```
kUartIrThreeSixteenthsWidth = 0U,
kUartIrOneSixteenthWidth = 1U,
kUartIrOneThirtysecondsWidth = 2U,
kUartIrOneFourthWidth = 3U }
```

UART infrared transmitter pulse width options.

- enum `uart_status_flag_t` {

kUartTxDataRegEmpty = 0U << UART_SHIFT | BP_UART_S1_TDRE,
 kUartTxComplete = 0U << UART_SHIFT | BP_UART_S1_TC,
 kUartRxDataRegFull = 0U << UART_SHIFT | BP_UART_S1_RDRF,
 kUartIdleLineDetect = 0U << UART_SHIFT | BP_UART_S1_IDLE,
 kUartRxOverrun = 0U << UART_SHIFT | BP_UART_S1_OR,
 kUartNoiseDetect = 0U << UART_SHIFT | BP_UART_S1_NF,
 kUartFrameErr = 0U << UART_SHIFT | BP_UART_S1_FE,
 kUartParityErr = 0U << UART_SHIFT | BP_UART_S1_PF,
 kUartLineBreakDetect = 1U << UART_SHIFT | BP_UART_S2_LBKDIF,
 kUartRxActiveEdgeDetect = 1U << UART_SHIFT | BP_UART_S2_RXEDGIF,
 kUartRxActive = 1U << UART_SHIFT | BP_UART_S2_RAF }

UART status flags.

- enum `uart_interrupt_t` {

kUartIntLinBreakDetect = 0U << UART_SHIFT | BP_UART_BDH_LBKDIE,
 kUartIntRxActiveEdge = 0U << UART_SHIFT | BP_UART_BDH_RXEDGIE,
 kUartIntTxDataRegEmpty = 1U << UART_SHIFT | BP_UART_C2_TIE,
 kUartIntTxComplete = 1U << UART_SHIFT | BP_UART_C2_TCIE,
 kUartIntRxDataRegFull = 1U << UART_SHIFT | BP_UART_C2_RIE,
 kUartIntIdleLine = 1U << UART_SHIFT | BP_UART_C2_ILIE,
 kUartIntRxOverrun = 2U << UART_SHIFT | BP_UART_C3_ORIE,
 kUartIntNoiseErrFlag = 2U << UART_SHIFT | BP_UART_C3_NEIE,
 kUartIntFrameErrFlag = 2U << UART_SHIFT | BP_UART_C3_FEIE,
 kUartIntParityErrFlag = 2U << UART_SHIFT | BP_UART_C3_PEIE }

UART interrupt configuration structure, default settings are 0 (disabled).

UART Common Configurations

- void `UART_HAL_Init` (uint32_t baseAddr)

Initializes the UART controller.
- static void `UART_HAL_EnableTransmitter` (uint32_t baseAddr)

Enables the UART transmitter.
- static void `UART_HAL_DisableTransmitter` (uint32_t baseAddr)

Disables the UART transmitter.
- static bool `UART_HAL_IsTransmitterEnabled` (uint32_t baseAddr)

Gets the UART transmitter enabled/disabled configuration setting.
- static void `UART_HAL_EnableReceiver` (uint32_t baseAddr)

Enables the UART receiver.
- static void `UART_HAL_DisableReceiver` (uint32_t baseAddr)

Disables the UART receiver.
- static bool `UART_HAL_IsReceiverEnabled` (uint32_t baseAddr)

UART HAL driver

- Gets the UART receiver enabled/disabled configuration setting.
- `uart_status_t UART_HAL_SetBaudRate` (uint32_t baseAddr, uint32_t sourceClockInHz, uint32_t baudRate)
Configures the UART baud rate.
- `void UART_HAL_SetBaudRateDivisor` (uint32_t baseAddr, uint16_t baudRateDivisor)
Sets the UART baud rate modulo divisor value.
- `static void UART_HAL_SetBitCountPerChar` (uint32_t baseAddr, `uart_bit_count_per_char_t` bitCountPerChar)
Configures the number of bits per character in the UART controller.
- `static void UART_HAL_SetParityMode` (uint32_t baseAddr, `uart_parity_mode_t` parityMode)
Configures the parity mode in the UART controller.
- `void UART_HAL_SetTxRxInversionCmd` (uint32_t baseAddr, bool rxInvertEnable, bool txInvertEnable)
Configures the transmit and receive inversion control in UART controller.

UART Interrupts and DMA

- `void UART_HAL_SetIntMode` (uint32_t baseAddr, `uart_interrupt_t` interrupt, bool enable)
Configures the UART module interrupts to enable/disable various interrupt sources.
- `bool UART_HAL_GetIntMode` (uint32_t baseAddr, `uart_interrupt_t` interrupt)
Returns whether the UART module interrupts is enabled/disabled.
- `static void UART_HAL_SetTxDataRegEmptyIntCmd` (uint32_t baseAddr, bool enable)
Enables or disables the tx_data_register_empty_interrupt.
- `static bool UART_HAL_GetTxDataRegEmptyIntCmd` (uint32_t baseAddr)
Gets the configuration of the tx_data_register_empty_interrupt enable setting.
- `static void UART_HAL_SetRxDataRegFullIntCmd` (uint32_t baseAddr, bool enable)
Disables the rx_data_register_full_interrupt.
- `static bool UART_HAL_GetRxDataRegFullIntCmd` (uint32_t baseAddr)
Gets the configuration of the rx_data_register_full_interrupt enable setting.
- `void UART_HAL_ConfigureDma` (uint32_t baseAddr, bool txDmaConfig, bool rxDmaConfig)
Configures the UART DMA requests for the Transmitter and Receiver.
- `bool UART_HAL_IsTxdmaEnabled` (uint32_t baseAddr)
Gets the UART Transmit DMA request configuration setting.
- `bool UART_HAL_IsRxdmaEnabled` (uint32_t baseAddr)
Gets the UART Receive DMA request configuration setting.
- `static uint32_t UART_HAL_GetDataRegAddr` (uint32_t baseAddr)
Get UART tx/rx data register address.

UART Transfer Functions

- `void UART_HAL_Putchar` (uint32_t baseAddr, uint8_t data)
This function allows the user to send an 8-bit character from the UART data register.
- `void UART_HAL_Putchar9` (uint32_t baseAddr, uint16_t data)
This function allows the user to send a 9-bit character from the UART data register.
- `void UART_HAL_Getchar` (uint32_t baseAddr, uint8_t *readData)
This function gets a received 8-bit character from the UART data register.
- `void UART_HAL_Getchar9` (uint32_t baseAddr, uint16_t *readData)
This function gets a received 9-bit character from the UART data register.

UART Special Feature Configurations

- static void [UART_HAL_SetWaitModeOperation](#) (uint32_t baseAddr, [uart_operation_config_t](#) mode)

Configures the UART to either operate or cease to operate in WAIT mode.
- static [uart_operation_config_t](#) [UART_HAL_GetWaitModeOperation](#) (uint32_t baseAddr)

Determines if the UART operates or ceases to operate in WAIT mode.
- static void [UART_HAL_SetLoopCmd](#) (uint32_t baseAddr, bool enable)

Configures the UART loopback operation.
- static void [UART_HAL_SetReceiverSource](#) (uint32_t baseAddr, [uart_receiver_source_t](#) source)

Configures the UART single-wire operation.
- static void [UART_HAL_SetTransmitterDir](#) (uint32_t baseAddr, [uart_singlewire_txdir_t](#) direction)

Configures the UART transmit direction while in single-wire mode.
- [uart_status_t](#) [UART_HAL_PutReceiverInStandbyMode](#) (uint32_t baseAddr)

Places the UART receiver in standby mode.
- static void [UART_HAL_PutReceiverInNormalMode](#) (uint32_t baseAddr)

Places the UART receiver in normal mode (disable standby mode operation).
- static bool [UART_HAL_IsReceiverInStandby](#) (uint32_t baseAddr)

Determines if the UART receiver is currently in standby mode.
- static void [UART_HAL_SetReceiverWakeupMethod](#) (uint32_t baseAddr, [uart_wakeup_method_t](#) method)

Selects the UART receiver wakeup method (idle-line or address-mark) from standby mode.
- static [uart_wakeup_method_t](#) [UART_HAL_GetReceiverWakeupMethod](#) (uint32_t baseAddr)

Gets the UART receiver wakeup method (idle-line or address-mark) from standby mode.
- void [UART_HAL_ConfigIdleLineDetect](#) (uint32_t baseAddr, uint8_t idleLine, uint8_t rxWakeIdleDetect)

Configures the operation options of the UART idle line detect.
- static void [UART_HAL_SetBreakCharTransmitLength](#) (uint32_t baseAddr, [uart_break_char_length_t](#) length)

Configures the UART break character transmit length.
- static void [UART_HAL_SetBreakCharDetectLength](#) (uint32_t baseAddr, [uart_break_char_length_t](#) length)

Configures the UART break character detect length.
- static void [UART_HAL_SetBreakCharCmd](#) (uint32_t baseAddr, bool enable)

Configures the UART transmit send break character operation.
- void [UART_HAL_SetMatchAddress](#) (uint32_t baseAddr, bool matchAddrMode1, bool matchAddrMode2, uint8_t matchAddrValue1, uint8_t matchAddrValue2)

Configures the UART match address mode control operation.

UART Status Flags

- bool [UART_HAL_GetStatusFlag](#) (uint32_t baseAddr, [uart_status_flag_t](#) statusFlag)

Gets all UART status flag states.
- static bool [UART_HAL_IsTxDataRegEmpty](#) (uint32_t baseAddr)

Gets the UART Transmit data register empty flag.
- static bool [UART_HAL_IsTxComplete](#) (uint32_t baseAddr)

Gets the UART Transmission complete flag.
- static bool [UART_HAL_IsRxDataRegFull](#) (uint32_t baseAddr)

Gets the UART Receive data register full flag.

UART HAL driver

- `uart_status_t UART_HAL_ClearStatusFlag (uint32_t baseAddr, uart_status_flag_t statusFlag)`
Clears an individual and specific UART status flag.
- `void UART_HAL_ClearAllNonAutoclearStatusFlags (uint32_t baseAddr)`
Clears all UART status flags.

25.1.1 Enumeration Type Documentation

25.1.1.1 enum `uart_status_t`

25.1.1.2 enum `uart_stop_bit_count_t`

These constants define the number of allowable stop bits to configure in a UART baseAddr.

Enumerator

kUartOneStopBit one stop bit

kUartTwoStopBit two stop bits

25.1.1.3 enum `uart_parity_mode_t`

These constants define the UART parity mode options: disabled or enabled of type even or odd.

Enumerator

kUartParityDisabled parity disabled

kUartParityEven parity enabled, type even, bit setting: PE|PT = 10

kUartParityOdd parity enabled, type odd, bit setting: PE|PT = 11

25.1.1.4 enum `uart_bit_count_per_char_t`

These constants define the number of allowable data bits per UART character. Note, check the UART documentation to determine if the desired UART baseAddr supports the desired number of data bits per UART character.

Enumerator

kUart8BitsPerChar 8-bit data characters

kUart9BitsPerChar 9-bit data characters

25.1.1.5 enum `uart_operation_config_t`

This provides constants for UART operational states: "operates normally" or "stops/ceases operation"

Enumerator

kUartOperates UART continues to operate normally.

kUartStops UART ceases operation.

25.1.1.6 enum uart_receiver_source_t

Enumerator

kUartLoopBack Internal loop back mode.

kUartSingleWire Single wire mode.

25.1.1.7 enum uart_wakeup_method_t

This provides constants for the two UART wakeup methods: idle-line or address-mark.

Enumerator

kUartIdleLineWake The idle-line wakes UART receiver from standby.

kUartAddrMarkWake The address-mark wakes UART receiver from standby.

25.1.1.8 enum uart_idle_line_select_t

This provides constants for the UART idle character bit-count start: either after start or stop bit.

Enumerator

kUartIdleLineAfterStartBit UART idle character bit count start after start bit.

kUartIdleLineAfterStopBit UART idle character bit count start after stop bit.

25.1.1.9 enum uart_break_char_length_t

This provides constants for the UART break character length for both transmission and detection purposes. Note that the actual maximum bit times may vary depending on the UART baseAddr.

Enumerator

kUartBreakChar10BitMinimum UART break char length 10 bit times (if M = 0, SBNS = 0) or 11 (if M = 1, SBNS = 0 or M = 0, SBNS = 1) or 12 (if M = 1, SBNS = 1 or M10 = 1, SNBS = 0) or 13 (if M10 = 1, SNBS = 1)

kUartBreakChar13BitMinimum UART break char length 13 bit times (if M = 0, SBNS = 0) or 14 (if M = 1, SBNS = 0 or M = 0, SBNS = 1) or 15 (if M = 1, SBNS = 1 or M10 = 1, SNBS = 0) or 16 (if M10 = 1, SNBS = 1)

25.1.1.10 enum uart_singlewire_txdirection_t

This provides constants for the UART transmit direction when configured for single-wire mode. The transmit line TXDIR is either an input or output.

Enumerator

kUartSinglewireTxdirIn UART Single-Wire mode TXDIR input.

kUartSinglewireTxdirOut UART Single-Wire mode TXDIR output.

25.1.1.11 enum uart_ir_tx_pulsewidth_t

This provides constants for the UART infrared (IR) pulse widths. Options include 3/16, 1/16 1/32, and 1/4 pulse widths.

Enumerator

kUartIrThreeSixteenthsWidth 3/16 pulse

kUartIrOneSixteenthWidth 1/16 pulse

kUartIrOneThirtysecondsWidth 1/32 pulse

kUartIrOneFourthWidth 1/4 pulse

25.1.1.12 enum uart_status_flag_t

This provides constants for the UART status flags for use in the UART functions.

Enumerator

kUartTxDataRegEmpty Tx data register empty flag, sets when Tx buffer is empty.

kUartTxComplete Transmission complete flag, sets when transmission activity complete.

kUartRxDataRegFull Rx data register full flag, sets when the receive data buffer is full.

kUartIdleLineDetect Idle line detect flag, sets when idle line detected.

kUartRxOverrun Rrx Overrun, sets when new data is received before data is read from receive register.

kUartNoiseDetect Rrx takes 3 samples of each received bit. If any of these samples differ, noise flag sets

kUartFrameErr Frame error flag, sets if logic 0 was detected where stop bit expected.

kUartParityErr If parity enabled, sets upon parity error detection.

kUartLineBreakDetect LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.

kUartRxActiveEdgeDetect Rx pin active edge interrupt flag, sets when active edge detected.

kUartRxActive Receiver Active Flag (RAF), sets at beginning of valid start bit.

25.1.1.13 enum uart_interrupt_t

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

Enumerator

- kUartIntLinBreakDetect* LIN break detect.
- kUartIntRxActiveEdge* RX Active Edge.
- kUartIntTxDataRegEmpty* Transmit data register empty.
- kUartIntTxComplete* Transmission complete.
- kUartIntRxDataRegFull* Receiver data register full.
- kUartIntIdleLine* Idle line.
- kUartIntRxOverrun* Receiver Overrun.
- kUartIntNoiseErrFlag* Noise error flag.
- kUartIntFrameErrFlag* Framing error flag.
- kUartIntParityErrFlag* Parity error flag.

25.1.2 Function Documentation

25.1.2.1 void UART_HAL_Init(uint32_t baseAddr)

This function initializes the module to a known state.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.1.2.2 static void UART_HAL_EnableTransmitter(uint32_t baseAddr) [inline], [static]

This function allows the user to enable the UART transmitter.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.1.2.3 static void UART_HAL_DisableTransmitter(uint32_t baseAddr) [inline], [static]

This function allows the user to disable the UART transmitter.

UART HAL driver

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.1.2.4 static bool UART_HAL_IsTransmitterEnabled (uint32_t *baseAddr*) [inline], [static]

This function allows the user to get the setting of the UART transmitter.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The state of UART transmitter enable(true)/disable(false) setting.

25.1.2.5 static void UART_HAL_EnableReceiver (uint32_t *baseAddr*) [inline], [static]

This function allows the user to enable the UART receiver.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.1.2.6 static void UART_HAL_DisableReceiver (uint32_t *baseAddr*) [inline], [static]

This function allows the user to disable the UART receiver.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.1.2.7 static bool UART_HAL_IsReceiverEnabled (uint32_t *baseAddr*) [inline], [static]

This function allows the user to get the setting of the UART receiver.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The state of UART receiver enable(true)/disable(false) setting.

25.1.2.8 **uart_status_t UART_HAL_SetBaudRate (uint32_t *baseAddr*, uint32_t *sourceClockInHz*, uint32_t *baudRate*)**

This function programs the UART baud rate to the desired value passed in by the user. The user must also pass in the module source clock so that the function can calculate the baud rate divisors to their appropriate values. In some UART baseAddrs it is required that the transmitter/receiver be disabled before calling this function. Generally this is applied to all UARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	UART module base address.
<i>sourceClockInHz</i>	UART source input clock in Hz.
<i>baudRate</i>	UART desired baud rate.

Returns

An error code or kStatus_UART_Success

25.1.2.9 **void UART_HAL_SetBaudRateDivisor (uint32_t *baseAddr*, uint16_t *baudRateDivisor*)**

This function allows the user to program the baud rate divisor directly in situations where the divisor value is known. In this case, the user may not want to call the [UART_HAL_SetBaudRate\(\)](#) function, as the divisor is already known.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

UART HAL driver

<i>baudRate-Divisor</i>	The baud rate modulo division "SBR" value.
-------------------------	--

25.1.2.10 static void UART_HAL_SetBitCountPerChar (uint32_t *baseAddr*, uart_bit_count_per_char_t *bitCountPerChar*) [inline], [static]

This function allows the user to configure the number of bits per character according to the typedef uart_bit_count_per_char_t.

Parameters

<i>baseAddr</i>	UART module base address.
<i>bitCountPerChar</i>	Number of bits per char (8, 9, or 10, depending on the UART baseAddr).

25.1.2.11 static void UART_HAL_SetParityMode (uint32_t *baseAddr*, uart_parity_mode_t *parityMode*) [inline], [static]

This function allows the user to configure the parity mode of the UART controller to disable it or enable it for even parity or for odd parity.

Parameters

<i>baseAddr</i>	UART module base address.
<i>parityMode</i>	Parity mode setting (enabled, disable, odd, even - see parity_mode_t struct).

25.1.2.12 void UART_HAL_SetTxRxInversionCmd (uint32_t *baseAddr*, bool *rxInvertEnable*, bool *txInvertEnable*)

This function allows the user to invert the transmit and receive signals, independently. This function should only be called when the UART is between transmit and receive packets.

Parameters

<i>baseAddr</i>	UART module base address.
<i>rxInvert</i>	Enable (true) or disable (false) receive inversion.
<i>txInvert</i>	Enable (true) or disable (false) transmit inversion.

25.1.2.13 **void UART_HAL_SetIntMode (uint32_t *baseAddr*, uart_interrupt_t *interrupt*, bool *enable*)**

UART HAL driver

Parameters

<i>baseAddr</i>	UART module base address.
<i>interrupt</i>	UART interrupt configuration data.
<i>enable</i>	true: enable, false: disable.

25.1.2.14 **bool UART_HAL_GetIntMode (uint32_t *baseAddr*, uart_interrupt_t *interrupt*)**

Parameters

<i>baseAddr</i>	UART module base address.
<i>interrupt</i>	UART interrupt configuration data.

Returns

true: enable, false: disable.

25.1.2.15 **static void UART_HAL_SetTxDataRegEmptyIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]**

Parameters

<i>baseAddr</i>	UART module base address.
<i>enable</i>	true: enable, false: disable.

25.1.2.16 **static bool UART_HAL_GetTxDataRegEmptyIntCmd (uint32_t *baseAddr*) [inline], [static]**

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

setting of the interrupt enable bit.

25.1.2.17 **static void UART_HAL_SetRxDataRegFullIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]**

Parameters

<i>baseAddr</i>	UART module base address.
<i>enable</i>	true: enable, false: disable.

25.1.2.18 static bool UART_HAL_GetRxDataRegFullIntCmd (uint32_t *baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

Bit setting of the interrupt enable bit.

25.1.2.19 void UART_HAL_ConfigureDma (uint32_t *baseAddr*, bool *txDmaConfig*, bool *rxDmaConfig*)

This function allows the user to configure the transmit data register empty flag to generate an interrupt request (default) or a DMA request. Similarly, this function allows the user to configure the receive data register full flag to generate an interrupt request (default) or a DMA request.

Parameters

<i>baseAddr</i>	UART module base address.
<i>txDmaConfig</i>	Transmit DMA request configuration setting (enable: true /disable: false).
<i>rxDmaConfig</i>	Receive DMA request configuration setting (enable: true/disable: false).

25.1.2.20 bool UART_HAL_IsTxdmaEnabled (uint32_t *baseAddr*)

This function returns the configuration setting of the Transmit DMA request.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

Transmit DMA request configuration setting (enable: true /disable: false).

25.1.2.21 **bool UART_HAL_IsRxdmaEnabled (uint32_t *baseAddr*)**

This function returns the configuration setting of the Receive DMA request.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

Receive DMA request configuration setting (enable: true / disable: false).

25.1.2.22 static uint32_t UART_HAL_GetDataRegAddr (uint32_t *baseAddr*) [inline], [static]

This function is used for DMA transfer.

Returns

UART tx/rx data register address.

25.1.2.23 void UART_HAL_Putchar (uint32_t *baseAddr*, uint8_t *data*)

Parameters

<i>baseAddr</i>	UART module base address.
<i>data</i>	The data to send of size 8-bit.

25.1.2.24 void UART_HAL_Putchar9 (uint32_t *baseAddr*, uint16_t *data*)

Parameters

<i>baseAddr</i>	UART module base address.
<i>data</i>	The data to send of size 9-bit.

25.1.2.25 void UART_HAL_Getchar (uint32_t *baseAddr*, uint8_t * *readData*)

Parameters

UART HAL driver

<i>baseAddr</i>	UART module base address.
<i>readData</i>	The received data read from data register of size 8-bit.

25.1.2.26 void **UART_HAL_Getchar9** (**uint32_t baseAddr, uint16_t * readData**)

Parameters

<i>baseAddr</i>	UART module base address.
<i>readData</i>	The received data read from data register of size 9-bit.

25.1.2.27 static void **UART_HAL_SetWaitModeOperation** (**uint32_t baseAddr, uart_operation_config_t mode**) [inline], [static]

The function configures the UART to either operate or cease to operate when WAIT mode is entered.

Parameters

<i>baseAddr</i>	UART module base address.
<i>mode</i>	The UART WAIT mode operation - operates or ceases to operate in WAIT mode.

25.1.2.28 static **uart_operation_config_t** **UART_HAL_GetWaitModeOperation** (**uint32_t baseAddr**) [inline], [static]

This function returns kUartOperates if the UART has been configured to operate in WAIT mode. Else it returns KUartStops if the UART has been configured to cease-to-operate in WAIT mode.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The UART WAIT mode operation configuration, returns either kUartOperates or KUartStops.

25.1.2.29 static void **UART_HAL_SetLoopCmd** (**uint32_t baseAddr, bool enable**) [inline], [static]

This function enables or disables the UART loopback operation.

Parameters

<i>baseAddr</i>	UART module base address.
<i>enable</i>	The UART loopback mode configuration, either disabled (false) or enabled (true).

25.1.2.30 static void **UART_HAL_SetReceiverSource** (**uint32_t baseAddr**, **uart_receiver_source_t source**) [**inline**], [**static**]

This function enables or disables the UART single-wire operation. In some UART baseAddrs it is required that the transmitter/receiver be disabled before calling this function. This may be applied to all UARTs to ensure safe operation.

Parameters

<i>baseAddr</i>	UART module base address.
<i>enable</i>	The UART single-wire mode configuration, either disabled (false) or enabled (true).

25.1.2.31 static void **UART_HAL_SetTransmitterDir** (**uint32_t baseAddr**, **uart_singlewire_txdir_t direction**) [**inline**], [**static**]

This function configures the transmitter direction when the UART is configured for single-wire operation.

Parameters

<i>baseAddr</i>	UART module base address.
<i>direction</i>	The UART single-wire mode transmit direction configuration of type <code>uart_singlewire_txdir_t</code> (either <code>kUartSinglewireTxdirIn</code> or <code>kUartSinglewireTxdirOut</code>).

25.1.2.32 **uart_status_t** **UART_HAL_PutReceiverInStandbyMode** (**uint32_t baseAddr**)

This function, when called, places the UART receiver into standby mode. In some UART baseAddrs, there are conditions that must be met before placing Rx in standby mode. Before placing UART in standby, determine if receiver is set to wake on idle, and if receiver is already in idle state. NOTE: RWU should only be set with C1[WAKE] = 0 (wakeup on idle) if the channel is currently not idle. This can be determined by the S2[RAF] flag. If set to wake up FROM an IDLE event and the channel is already idle, it is possible that the UART will discard data because data must be received (or a LIN break detect) after an IDLE is detected before IDLE is allowed to be reasserted.

UART HAL driver

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

Error code or kStatus_UART_Success.

25.1.2.33 static void UART_HAL_PutReceiverInNormalMode (uint32_t *baseAddr*) [inline], [static]

This function, when called, places the UART receiver into normal mode and out of standby mode.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.1.2.34 static bool UART_HAL_IsReceiverInStandby (uint32_t *baseAddr*) [inline], [static]

This function determines the state of the UART receiver. If it returns true, this means that the UART receiver is in standby mode; if it returns false, the UART receiver is in normal mode.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The UART receiver is in normal mode (false) or standby mode (true).

25.1.2.35 static void UART_HAL_SetReceiverWakeupMethod (uint32_t *baseAddr*, uart_wakeup_method_t *method*) [inline], [static]

This function configures the wakeup method of the UART receiver from standby mode. The options are idle-line wake or address-mark wake.

Parameters

<i>baseAddr</i>	UART module base address.
<i>method</i>	The UART receiver wakeup method options: kUartIdleLineWake - Idle-line wake or kUartAddrMarkWake - address-mark wake.

25.1.2.36 static uart_wakeup_method_t UART_HAL_GetReceiverWakeupMethod (uint32_t *baseAddr*) [inline], [static]

This function returns how the UART receiver is configured to wake from standby mode. The wake method options that can be returned are kUartIdleLineWake or kUartAddrMarkWake.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The UART receiver wakeup from standby method, false: kUartIdleLineWake (idle-line wake) or true: kUartAddrMarkWake (address-mark wake).

25.1.2.37 void UART_HAL_ConfigIdleLineDetect (uint32_t *baseAddr*, uint8_t *idleLine*, uint8_t *rxWakeIdleDetect*)

This function allows the user to configure the UART idle-line detect operation. There are two separate operations for the user to configure, the idle line bit-count start and the receive wake up affect on IDLE status bit. The user will pass in a structure of type *uart_idle_line_config_t*.

Parameters

<i>baseAddr</i>	UART module base address.
<i>idleLine</i>	Idle bit count start: 0 - after start bit (default), 1 - after stop bit
<i>rxWakeIdle-Detect</i>	Receiver Wake Up Idle Detect. IDLE status bit operation during receive standby. Controls whether idle character that wakes up receiver will also set IDLE status bit. 0 - IDLE status bit doesn't get set (default), 1 - IDLE status bit gets set

25.1.2.38 static void UART_HAL_SetBreakCharTransmitLength (uint32_t *baseAddr*, uart_break_char_length_t *length*) [inline], [static]

This function allows the user to configure the UART break character transmit length. Refer to the typedef *uart_break_char_length_t* for setting options. In some UART baseAddrs it is required that the transmitter be disabled before calling this function. This may be applied to all UARTs to ensure safe operation.

UART HAL driver

Parameters

<i>baseAddr</i>	UART module base address.
<i>length</i>	The UART break character length setting of type <code>uart_break_char_length_t</code> , either a minimum 10-bit times or a minimum 13-bit times.

25.1.2.39 static void `UART_HAL_SetBreakCharDetectLength` (`uint32_t baseAddr, uart_break_char_length_t length`) [inline], [static]

This function allows the user to configure the UART break character detect length. Refer to the typedef `uart_break_char_length_t` for setting options.

Parameters

<i>baseAddr</i>	UART module base address.
<i>length</i>	The UART break character length setting of type <code>uart_break_char_length_t</code> , either a minimum 10-bit times or a minimum 13-bit times.

25.1.2.40 static void `UART_HAL_SetBreakCharCmd` (`uint32_t baseAddr, bool enable`) [inline], [static]

This function allows the user to queue a UART break character to send. If true is passed into the function, then a break character is queued for transmission. A break character will continuously be queued until this function is called again when a false is passed into this function.

Parameters

<i>baseAddr</i>	UART module base address.
<i>enable</i>	If false, the UART normal/queue break character setting is disabled, which configures the UART for normal transmitter operation. If true, a break character is queued for transmission.

25.1.2.41 void `UART_HAL_SetMatchAddress` (`uint32_t baseAddr, bool matchAddrMode1, bool matchAddrMode2, uint8_t matchAddrValue1, uint8_t matchAddrValue2`)

(Note: Feature available on select UART baseAddrs)

The function allows the user to configure the UART match address control operation. The user has the option to enable the match address mode and to program the match address value. There are two match address modes, each with its own enable and programmable match address value.

Parameters

<i>baseAddr</i>	UART module base address.
<i>matchAddr-Mode1</i>	If true, this enables match address mode 1 (MAEN1), where false disables.
<i>matchAddr-Mode2</i>	If true, this enables match address mode 2 (MAEN2), where false disables.
<i>matchAddr-Value1</i>	The match address value to program for match address mode 1.
<i>matchAddr-Value2</i>	The match address value to program for match address mode 2.

25.1.2.42 **bool UART_HAL_GetStatusFlag (uint32_t *baseAddr*, uart_status_flag_t *statusFlag*)**

Parameters

<i>baseAddr</i>	UART module base address.
<i>statusFlag</i>	Status flag name.

25.1.2.43 **static bool UART_HAL_IsTxDataRegEmpty (uint32_t *baseAddr*) [inline], [static]**

This function returns the state of the UART Transmit data register empty flag.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The status of Transmit data register empty flag, which is set when transmit buffer is empty.

25.1.2.44 **static bool UART_HAL_IsTxComplete (uint32_t *baseAddr*) [inline], [static]**

This function returns the state of the UART Transmission complete flag.

UART HAL driver

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The status of Transmission complete flag, which is set when the transmitter is idle (transmission activity complete).

25.1.2.45 static bool UART_HAL_IsRxDataRegFull (uint32_t *baseAddr*) [inline], [static]

This function returns the state of the UART Receive data register full flag.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

Returns

The status of Receive data register full flag, which is set when the receive data buffer is full.

25.1.2.46 uart_status_t UART_HAL_ClearStatusFlag (uint32_t *baseAddr*, uart_status_flag_t *statusFlag*)

This function allows the user to clear an individual and specific UART status flag. Refer to structure definition `uart_status_flag_t` for list of status bits.

Parameters

<i>baseAddr</i>	UART module base address.
<i>statusFlag</i>	The desired UART status flag to clear.

Returns

An error code or `kStatus_UART_Success`.

25.1.2.47 void UART_HAL_ClearAllNonAutoclearStatusFlags (uint32_t *baseAddr*)

This function tries to clear all of the UART status flags. In some cases, some of the status flags may not get cleared because the condition that set the flag may still exist.

Parameters

<i>baseAddr</i>	UART module base address.
-----------------	---------------------------

25.2 UART Peripheral Driver

This chapter describes the programming interface of the UART Peripheral driver.

Data Structures

- struct `uart_state_t`
Runtime state of the UART driver. [More...](#)
- struct `uart_user_config_t`
User configuration structure for the UART driver. [More...](#)

TypeDefs

- typedef `uart_status_t(* uart_rx_callback_t)(uint8_t *rxByte, void *param)`
UART receive callback function type.

UART Driver

- `uart_status_t UART_DRV_Init` (`uint32_t` instance, `uart_state_t *uartStatePtr`, const `uart_user_config_t *uartUserConfig`)
Initializes a UART instance for operation.
- `void UART_DRV_Deinit` (`uint32_t` instance)
Shuts down the UART by disabling interrupts and the transmitter/receiver.
- `uart_rx_callback_t UART_DRV_InstallRxCallback` (`uint32_t` instance, `uart_rx_callback_t` function, `void *callbackParam`)
Installs callback function for the UART receive.
- `uart_status_t UART_DRV_SendDataBlocking` (`uint32_t` instance, const `uint8_t *txBuff`, `uint32_t txSize`, `uint32_t timeout`)
Sends (transmits) data out through the UART module using a blocking method.
- `uart_status_t UART_DRV_SendData` (`uint32_t` instance, const `uint8_t *txBuff`, `uint32_t txSize`)
Sends (transmits) data through the UART module using a non-blocking method.
- `uart_status_t UART_DRV_GetTransmitStatus` (`uint32_t` instance, `uint32_t *bytesRemaining`)
Returns whether the previous UART transmit has finished.
- `uart_status_t UART_DRV_AbortSendingData` (`uint32_t` instance)
Terminates an asynchronous UART transmission early.
- `uart_status_t UART_DRV_ReceiveDataBlocking` (`uint32_t` instance, `uint8_t *rxBuff`, `uint32_t rxSize`, `uint32_t timeout`)
Gets (receives) data from the UART module using a blocking method.
- `uart_status_t UART_DRV_ReceiveData` (`uint32_t` instance, `uint8_t *rxBuff`, `uint32_t rxSize`)
Gets (receives) data from the UART module using a non-blocking method.
- `uart_status_t UART_DRV_GetReceiveStatus` (`uint32_t` instance, `uint32_t *bytesRemaining`)
Returns whether the previous UART receive is complete.
- `uart_status_t UART_DRV_AbortReceivingData` (`uint32_t` instance)
Terminates an asynchronous UART receive early.

25.2.0.48 UART Peripheral Driver

Overview

The UART peripheral driver is used to transfer data to and from external devices on the Universal Asynchronous Receiver/Transmitter (UART) serial bus. It provides a way to transmit and receive buffers of data with a single function call.

Device structures

The driver uses an instantiation of the `uart_state_t` structure to maintain the current state of a particular UART instance module driver. This structure holds data that is used by the UART Peripheral driver to communicate between the transmit and receive transfer functions and the interrupt handler. The interrupt handler also uses this information to keep track of its progress. Because the driver itself does not statically allocate memory, the caller provides memory for the driver state structure during init. The user is only responsible to pass in the memory for this run-time state structure where the UART driver will take care of filling out the members.

User configuration structures

The UART driver uses instances of the user configuration structure `uart_user_config_t` for the UART driver. This enables configuration of the most common settings of the UART with a single function call. Settings include: UART baud rate; UART parity mode: disabled (default), or even or odd; the number of stop bits; the number of bits per data word.

Initialization

To initialize the UART driver, call the `UART_DRV_Init()` function and pass the instance number of the UART peripheral you want to use, memory for the run-time state structure, and a pointer to the user configuration structure. For example, to use UART0 pass a value of 0 to the initialization function. Then, pass the memory for the run-time state structure and, finally, pass a user configuration structure of the type `uart_user_config_t` as shown here:

```
// UART configuration structure for user
typedef struct UartUserConfig {
    uint32_t baudRate;
    uart_parity_mode_t parityMode;
    uart_stop_bit_count_t stopBitCount;
    uart_bit_count_per_char_t bitCountPerChar;
} uart_user_config_t;
```

Typically, the `uart_user_config_t` instantiation is configured as 8-bit-char, no-parity, 1-stop-bit (8-n-1) with a 9600 bps baud rate. The `uart_user_config_t` instantiation can be easily modified to configure the UART

UART Peripheral Driver

Peripheral driver either to a different baud rate or character transfer features. This is an example code to set up a user UART configuration instantiation:

```
uart_user_config_t uartConfig;
uartConfig.baudRate = 9600;
uartConfig.bitCountPerChar = kUart8BitsPerChar;
uartConfig.parityMode = kUartParityDisabled;
uartConfig.stopBitCount = kUartOneStopBit;
```

This example shows how to make the [UART_DRV_Init\(\)](#) function call given the user configuration structure previously given, and the UART instance 0.

```
uint32_t uartInstance = 0;
uart_state_t uartState; // user provides memory for the driver state structure

UART_DRV_Init(uartInstance, &uartConfig, &uartState);
```

Transfers

The driver implements transmit and receive functions to transfer buffers of data. The driver also supports two different modes for transferring data: blocking and non-blocking.

The non-blocking transmit and receive functions are the [UART_DRV_SendData\(\)](#) and [UART_DRV_ReceiveData\(\)](#) functions.

The blocking (async) transmit and receive functions are the [UART_DRV_SendDataBlocking\(\)](#) and [UART_DRV_ReceiveDataBlocking\(\)](#) functions.

In all cases mentioned here, the functions are interrupt driven.

The following code examples show how to use the aforementioned functions. First, it is assumed that the UART module has been initialized as described previously in the Initialization chapter.

For blocking transfer functions transmit and receive:

```
uint8_t sourceBuff[26] = {0}; // sourceBuff can be filled out with desired data
uint8_t readBuffer[10] = {0}; // readBuffer gets filled with UART_DRV_ReceiveData function

uint32_t byteCount = sizeof(sourceBuff);
uint32_t rxRemainingSize = sizeof(readBuffer);

// for each use there, set timeout as "1"
// uartState is the run-time state. Pass in memory for this
// declared previously in the initialization chapter
UART_DRV_SendDataBlocking(&uartState, sourceBuff, byteCount, 1); // function won't
// return until transmit is complete
UART_DRV_ReceiveDataBlocking(&uartState, readBuffer, 1, timeoutValue); // function won't return until it receives all data
```

For non-blocking (async) transfer functions transmit and receive:

```
uint8_t *pTxBuff;
uint8_t rxBuff[10];
uint32_t txRemainingSize, rxRemainingSize;
```

```

// assume pTxBuff and txRemainingSize have been initialized
UART_DRV_SendData(&uartState, pTxBuff, txRemainingSize);

// now check on status of transmit and wait until done, the code can do something else and
// check back later, this is just an example
while (UART_DRV_GetTransmitStatus(&uartState, &bytesTransmittedCount) ==
    kStatus_UART_TxBusy);

// for receive, assume rxBuff is set up to receive data and rxRemainingSize is initialized
UART_DRV_ReceiveData(&uartState, rxBuff, rxRemainingSize);

// now check on status of receive and wait until done, the code can do something else and
// check back later, this is just an example
while (UART_DRV_GetReceiveStatus(&uartState, &bytesReceivedCount) ==
    kStatus_UART_RxBusy);

```

25.2.1 Data Structure Documentation

25.2.1.1 struct uart_state_t

This structure holds data that are used by the UART peripheral driver to communicate between the transfer function and the interrupt handler. The interrupt handler also uses this information to keep track of its progress. The user passes in the memory for the run-time state structure and the UART driver fills out the members.

Data Fields

- **uint8_t txFifoEntryCount**
Number of data word entries in TX FIFO.
- **const uint8_t * txBuff**
The buffer of data being sent.
- **uint8_t * rxBuff**
The buffer of received data.
- **volatile size_t txSize**
The remaining number of bytes to be transmitted.
- **volatile size_t rxSize**
The remaining number of bytes to be received.
- **volatile bool isTxBusy**
True if there is an active transmit.
- **volatile bool isRxBusy**
True if there is an active receive.
- **volatile bool isTxBlocking**
True if transmit is blocking transaction.
- **volatile bool isRxBlocking**
True if receive is blocking transaction.
- **semaphore_t txIrqSync**
Used to wait for ISR to complete its TX business.
- **semaphore_t rxIrqSync**
Used to wait for ISR to complete its RX business.
- **uart_rx_callback_t rxCallback**
Callback to invoke after receiving byte.

UART Peripheral Driver

- `void * rxCallbackParam`
Receive callback parameter pointer.

25.2.1.1.0.57 Field Documentation

25.2.1.1.0.57.1 `uint8_t uart_state_t::txFifoEntryCount`

25.2.1.1.0.57.2 `const uint8_t* uart_state_t::txBuff`

25.2.1.1.0.57.3 `uint8_t* uart_state_t::rxBuff`

25.2.1.1.0.57.4 `volatile size_t uart_state_t::txSize`

25.2.1.1.0.57.5 `volatile size_t uart_state_t::rxSize`

25.2.1.1.0.57.6 `volatile bool uart_state_t::isTxBusy`

25.2.1.1.0.57.7 `volatile bool uart_state_t::isRxBusy`

25.2.1.1.0.57.8 `volatile bool uart_state_t::isTxBlocking`

25.2.1.1.0.57.9 `volatile bool uart_state_t::isRxBlocking`

25.2.1.1.0.57.10 `semaphore_t uart_state_t::txIrqSync`

25.2.1.1.0.57.11 `semaphore_t uart_state_t::rxIrqSync`

25.2.1.1.0.57.12 `uart_rx_callback_t uart_state_t::rxCallback`

25.2.1.1.0.57.13 `void* uart_state_t::rxCallbackParam`

25.2.1.2 `struct uart_user_config_t`

Use an instance of this structure with the `UART_DRV_Init()` function. This enables configuration of the most common settings of the UART peripheral with a single function call. Settings include: UART baud rate; UART parity mode: disabled (default), or even or odd; the number of stop bits; the number of bits per data word.

Data Fields

- `uint32_t baudRate`
UART baud rate.
- `uart_parity_mode_t parityMode`
parity mode, disabled (default), even, odd
- `uart_stop_bit_count_t stopBitCount`
number of stop bits, 1 stop bit (default) or 2 stop bits
- `uart_bit_count_per_char_t bitCountPerChar`
number of bits, 8-bit (default) or 9-bit in a word (up to 10-bits in some UART instances)

25.2.2 Typedef Documentation

25.2.2.1 `typedef uart_status_t(* uart_rx_callback_t)(uint8_t *rxByte, void *param)`

25.2.3 Function Documentation

25.2.3.1 `uart_status_t UART_DRV_Init (uint32_t instance, uart_state_t * uartStatePtr, const uart_user_config_t * uartUserConfig)`

This function initializes the run-time state structure to keep track of the on-going transfers, ungates the clock to the UART module, initializes the module to user-defined settings and default settings, configures the IRQ state structure and enables the module-level interrupt to the core, and enables the UART module transmitter and receiver. This example shows how to set up the `uart_state_t` and the `uart_user_config_t` parameters and how to call the `UART_DRV_Init` function by passing in these parameters:

```
uart_user_config_t uartConfig;
uartConfig.baudRate = 9600;
uartConfig.bitCountPerChar = kUart8BitsPerChar;
uartConfig.parityMode = kUartParityDisabled;
uartConfig.stopBitCount = kUartOneStopBit;
uart_state_t uartState;
UART_DRV_Init(instance, &uartState, &uartConfig);
```

Parameters

<code>instance</code>	The UART instance number.
<code>uartStatePtr</code>	A pointer to the UART driver state structure memory. The user is only responsible to pass in the memory for this run-time state structure where the UART driver will take care of filling out the members. This run-time state structure keeps track of the current transfer in progress.
<code>uartUserConfig</code>	The user configuration structure of type <code>uart_user_config_t</code> . The user is responsible to fill out the members of this structure and to pass the pointer of this structure into this function.

Returns

An error code or `kStatus_UART_Success`.

25.2.3.2 `void UART_DRV_Deinit (uint32_t instance)`

This function disables the UART interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs).

UART Peripheral Driver

Parameters

<i>instance</i>	The UART instance number.
-----------------	---------------------------

25.2.3.3 **uart_rx_callback_t UART_DRV_InstallRxCallback (uint32_t *instance*, uart_rx_callback_t *function*, void * *callbackParam*)**

Parameters

<i>instance</i>	The UART instance number.
<i>function</i>	The UART receive callback function.
<i>callbackParam</i>	The UART receive callback parameter pointer.

Returns

Former UART receive callback function pointer.

25.2.3.4 **uart_status_t UART_DRV_SendDataBlocking (uint32_t *instance*, const uint8_t * *txBuff*, uint32_t *txSize*, uint32_t *timeout*)**

A blocking (also known as synchronous) function means that the function does not return until the transmit is complete. This blocking function is used to send data through the UART port.

Parameters

<i>instance</i>	The UART instance number.
<i>txBuff</i>	A pointer to the source buffer containing 8-bit data chars to send.
<i>txSize</i>	The number of bytes to send.
<i>timeout</i>	A timeout value for RTOS abstraction sync control in milliseconds (ms).

Returns

An error code or kStatus_UART_Success.

25.2.3.5 **uart_status_t UART_DRV_SendData (uint32_t *instance*, const uint8_t * *txBuff*, uint32_t *txSize*)**

A non-blocking (also known as synchronous) function means that the function returns immediately after initiating the transmit function. The application has to get the transmit status to see when the transmit is

complete. In other words, after calling non-blocking (asynchronous) send function, the application must get the transmit status to check if transmit is complete. The asynchronous method of transmitting and receiving allows the UART to perform a full duplex operation (simultaneously transmit and receive).

UART Peripheral Driver

Parameters

<i>instance</i>	The UART module base address.
<i>txBuff</i>	A pointer to the source buffer containing 8-bit data chars to send.
<i>txSize</i>	The number of bytes to send.

Returns

An error code or kStatus_UART_Success.

25.2.3.6 **uart_status_t UART_DRV_GetTransmitStatus (uint32_t *instance*, uint32_t * *bytesRemaining*)**

When performing an async transmit, call this function to ascertain the state of the current transmission: in progress (or busy) or complete (success). If the transmission is still in progress, the user can obtain the number of words that have been transferred.

Parameters

<i>instance</i>	The UART module base address.
<i>bytes-Remaining</i>	A pointer to a value that is filled in with the number of bytes that are remaining in the active transfer.

Return values

<i>kStatus_UART_Success</i>	The transmit has completed successfully.
<i>kStatus_UART_TxBusy</i>	The transmit is still in progress. <i>bytesTransmitted</i> is filled with the number of bytes which are transmitted up to that point.

25.2.3.7 **uart_status_t UART_DRV_AbortSendingData (uint32_t *instance*)**

During an async UART transmission, the user can terminate the transmission early if the transmission is still in progress.

Parameters

<i>instance</i>	The UART module base address.
-----------------	-------------------------------

Return values

<i>kStatus_UART_Success</i>	The transmit was successful.
<i>kStatus_UART_NoTransmitInProgress</i>	No transmission is currently in progress.

25.2.3.8 **uart_status_t UART_DRV_ReceiveDataBlocking (uint32_t *instance*, uint8_t * *rxBuff*, uint32_t *rxSize*, uint32_t *timeout*)**

A blocking (also known as synchronous) function means that the function does not return until the receive is complete. This blocking function sends data through the UART port.

Parameters

<i>instance</i>	The UART module base address.
<i>rxBuff</i>	A pointer to the buffer containing 8-bit read data chars received.
<i>rxSize</i>	The number of bytes to receive.
<i>timeout</i>	A timeout value for RTOS abstraction sync control in milliseconds (ms).

Returns

An error code or *kStatus_UART_Success*.

25.2.3.9 **uart_status_t UART_DRV_ReceiveData (uint32_t *instance*, uint8_t * *rxBuff*, uint32_t *rxSize*)**

A non-blocking (also known as synchronous) function means that the function returns immediately after initiating the receive function. The application has to get the receive status to see when the receive is complete. In other words, after calling non-blocking (asynchronous) get function, the application must get the receive status to check if receive is completed or not. The asynchronous method of transmitting and receiving allows the UART to perform a full duplex operation (simultaneously transmit and receive).

Parameters

<i>instance</i>	The UART module base address.
<i>rxBuff</i>	A pointer to the buffer containing 8-bit read data chars received.

UART Peripheral Driver

<i>rxSize</i>	The number of bytes to receive.
---------------	---------------------------------

Returns

An error code or kStatus_UART_Success.

25.2.3.10 **uart_status_t UART_DRV_GetReceiveStatus (uint32_t *instance*, uint32_t * *bytesRemaining*)**

When performing an async receive, the user can call this function to ascertain the state of the current receive progress: in progress (or busy) or complete (success). In addition, if the receive is still in progress, the user can obtain the number of words that have been currently received.

Parameters

<i>instance</i>	The UART module base address.
<i>bytesRemaining</i>	A pointer to a value that is filled in with the number of bytes which still need to be received in the active transfer.

Return values

<i>kStatus_UART_Success</i>	The receive has completed successfully.
<i>kStatus_UART_RxBusy</i>	The receive is still in progress. <i>bytesReceived</i> is filled with the number of bytes which are received up to that point.

25.2.3.11 **uart_status_t UART_DRV_AbortReceivingData (uint32_t *instance*)**

During an async UART receive, the user can terminate the receive early if the receive is still in progress.

Parameters

<i>instance</i>	The UART module base address.
-----------------	-------------------------------

Return values

<i>kStatus_UART_Success</i>	The receive was successful.
<i>kStatus_UART_NoTransmitInProgress</i>	No receive is currently in progress.

Chapter 26

Watchdog Timer (WDOG)

The Kinetis SDK provides both HAL and Peripheral drivers for the Watchdog Timer (WDOG) block of Kinetis devices.

Modules

- [WDOG HAL driver](#)

This part describes the programming interface of the WDOG HAL driver.

- [WDOG Peripheral Driver](#)

This part describes the programming interface of the WDOG Peripheral driver.

WDOG HAL driver

26.1 WDOG HAL driver

This chapter describes the programming interface of the WDOG HAL driver.

Data Structures

- union `wdog_common_config`
Define the common configure. [More...](#)

Enumerations

- enum `wdog_clock_source_t` {
 `kWdogClockSourceLpoClock` = 0x0U,
 `kWdogClockSourceBusClock` = 0x1U }
Watchdog clock source selection.
- enum `wdog_clock_prescaler_value_t` {
 `kWdogClockPrescalerValueDevide1` = 0x0U,
 `kWdogClockPrescalerValueDevide2` = 0x1U,
 `kWdogClockPrescalerValueDevide3` = 0x2U,
 `kWdogClockPrescalerValueDevide4` = 0x3U,
 `kWdogClockPrescalerValueDevide5` = 0x4U,
 `kWdogClockPrescalerValueDevide6` = 0x5U,
 `kWdogClockPrescalerValueDevide7` = 0x6U,
 `kWdogClockPrescalerValueDevide8` = 0x7U }
Define the selection of the clock prescaler.

Watchdog HAL.

- static void `WDOG_HAL_SetCommonConfig` (uint32_t baseAddr, `wdog_common_config` commonConfig)
Sets the WDOG common configure.
- static void `WDOG_HAL_Enable` (uint32_t baseAddr)
Enables the Watchdog module.
- static void `WDOG_HAL_Disable` (uint32_t baseAddr)
Disables the Watchdog module.
- static bool `WDOG_HAL_IsEnabled` (uint32_t baseAddr)
Checks whether the WDOG is enabled.
- static void `WDOG_HAL_SetIntCmd` (uint32_t baseAddr, bool enable)
Enables and disables the Watchdog interrupt.
- static bool `WDOG_HAL_GetIntCmd` (uint32_t baseAddr)
Checks whether the WDOG interrupt is enabled.
- static void `WDOG_HAL_SetClockSourceMode` (uint32_t baseAddr, `wdog_clock_source_t` clockSource)
Sets the Watchdog clock Source.
- static `wdog_clock_source_t` `WDOG_HAL_GetClockSourceMode` (uint32_t baseAddr)
Gets the Watchdog clock Source.

- static void **WDOG_HAL_SetWindowModeCmd** (uint32_t baseAddr, bool enable)

Enables and disables the Watchdog window mode.
- static bool **WDOG_HAL_GetWindowModeCmd** (uint32_t baseAddr)

Checks whether the window mode is enabled.
- static void **WDOG_HAL_SetRegisterUpdateCmd** (uint32_t baseAddr, bool enable)

Enables and disables the Watchdog write-once-only register update.
- static bool **WDOG_HAL_GetRegisterUpdateCmd** (uint32_t baseAddr)

Checks whether the register update is enabled.
- static void **WDOG_HAL_SetWorkInDebugModeCmd** (uint32_t baseAddr, bool enable)

Sets whether Watchdog is working while the CPU is in debug mode.
- static bool **WDOG_HAL_GetWorkInDebugModeCmd** (uint32_t baseAddr)

Checks whether the WDOG works while in the CPU debug mode.
- static void **WDOG_HAL_SetWorkInStopModeCmd** (uint32_t baseAddr, bool enable)

Sets whether the Watchdog is working while the CPU is in stop mode.
- static bool **WDOG_HAL_GetWorkInStopModeCmd** (uint32_t baseAddr)

Checks whether the WDOG works while in CPU stop mode.
- static void **WDOG_HAL_SetWorkInWaitModeCmd** (uint32_t baseAddr, bool enable)

Sets whether the Watchdog is working while the CPU is in wait mode.
- static bool **WDOG_HAL_GetWorkInWaitModeCmd** (uint32_t baseAddr)

Checks whether the WDOG works while in the CPU wait mode.
- static bool **WDOG_HAL_IsIntPending** (uint32_t baseAddr)

Gets the Watchdog interrupt status.
- static void **WDOG_HAL_ClearIntFlag** (uint32_t baseAddr)

Clears the Watchdog interrupt flag.
- static void **WDOG_HAL_SetTimeoutValue** (uint32_t baseAddr, uint32_t timeoutCount)

Set the Watchdog timeout value.
- static uint32_t **WDOG_HAL_GetTimeoutValue** (uint32_t baseAddr)

Gets the Watchdog timeout value.
- static uint32_t **WDOG_HAL_GetTimerOutputValue** (uint32_t baseAddr)

Gets the Watchdog timer output.
- static void **WDOG_HAL_SetClockPrescalerValueMode** (uint32_t baseAddr, **wdog_clock_prescaler_value_t** clockPrescaler)

Sets the Watchdog clock prescaler.
- static **wdog_clock_prescaler_value_t** **WDOG_HAL_GetClockPrescalerValueMode** (uint32_t baseAddr)

Gets the Watchdog clock prescaler.
- static void **WDOG_HAL_SetWindowValue** (uint32_t baseAddr, uint32_t windowValue)

Sets the Watchdog window value.
- static uint32_t **WDOG_HAL_GetWindowValue** (uint32_t baseAddr)

Gets the Watchdog window value.
- static void **WDOG_HAL_Unlock** (uint32_t baseAddr)

Unlocks the Watchdog register written.
- static void **WDOG_HAL_Refresh** (uint32_t baseAddr)

Refreshes the Watchdog timer.
- static void **WDOG_HAL_ResetSystem** (uint32_t baseAddr)

Resets the chip using the Watchdog.
- static uint32_t **WDOG_HAL_GetResetCount** (uint32_t baseAddr)

Gets the chip reset count that was reset by Watchdog.
- static void **WDOG_HAL_ClearResetCount** (uint32_t baseAddr)

Clears the chip reset count that was reset by Watchdog.
- void **WDOG_HAL_Init** (uint32_t baseAddr)

WDOG HAL driver

Restores the WDOG module to reset value.

26.1.1 Data Structure Documentation

26.1.1.1 union wdog_common_config

26.1.2 Enumeration Type Documentation

26.1.2.1 enum wdog_clock_source_t

Enumerator

kWdogClockSourceLpoClock Clock source is LPO clock.

kWdogClockSourceBusClock Clock source is Bus clock.

26.1.2.2 enum wdog_clock_prescaler_value_t

Enumerator

kWdogClockPrescalerValueDevide1 Divided by 1.

kWdogClockPrescalerValueDevide2 Divided by 2.

kWdogClockPrescalerValueDevide3 Divided by 3.

kWdogClockPrescalerValueDevide4 Divided by 4.

kWdogClockPrescalerValueDevide5 Divided by 5.

kWdogClockPrescalerValueDevide6 Divided by 6.

kWdogClockPrescalerValueDevide7 Divided by 7.

kWdogClockPrescalerValueDevide8 Divided by 8.

26.1.3 Function Documentation

26.1.3.1 static void WDOG_HAL_SetCommonConfig (uint32_t baseAddr, wdog_common_config commonConfig) [inline], [static]

This function is used to set the WDOG common configure. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock, the WCT window is still open and the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure that the WDOG_STCTRLH.ALLOW-UPDATE is 1 which means that the register update is enabled. The common configuration is controlled by the WDOG_STCTRLH. This is a write-once register and this interface is used to set all field of the WDOG_STCTRLH registers at the same time. If only one field needs to be set, the API can be used. These API write to the WDOG_STCTRLH register: [WDOG_HAL_Enable](#), [WDOG_HAL_Disable](#), [WDOG_HAL_SetIntCmd](#), [WDOG_HAL_SetClockSourceMode](#), [WDOG_HAL_SetWindowModeCmd](#), [WDOG_HAL_SetRegisterUpdateCmd](#), [WDOG_HAL_SetWorkInDebugModeCmd](#), [WDOG_HAL_SetWorkInStopModeCmd](#), [WDOG_HAL_SetWorkInWaitModeCmd](#)

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>commonConfig</i>	The common configure of the WDOG

26.1.3.2 static void WDOG_HAL_Enable(uint32_t *baseAddr*) [inline], [static]

This function enables the WDOG. Make sure that the WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.3 static void WDOG_HAL_Disable(uint32_t *baseAddr*) [inline], [static]

This function disables the WDOG. Make sure that the WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.4 static bool WDOG_HAL_IsEnabled(uint32_t *baseAddr*) [inline], [static]

This function checks whether the WDOG is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

WDOG HAL driver

Returns

false means WDOG is disabled, true means WODG is enabled.

26.1.3.5 static void WDOG_HAL_SetIntCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables or disables the WDOG interrupt. Make sure that the WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>enable</i>	false means disable watchdog interrupt and true means enable watchdog interrupt.

26.1.3.6 static bool WDOG_HAL_GetIntCmd (uint32_t *baseAddr*) [inline], [static]

This function checks whether the WDOG interrupt is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

false means interrupt is disabled, true means interrupt is enabled.

26.1.3.7 static void WDOG_HAL_SetClockSourceMode (uint32_t *baseAddr*, wdog_clock_source_t *clockSource*) [inline], [static]

This function sets the WDOG clock source. There are two clock sources that can be used: the LPO clock and the bus clock. Make sure that the WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>clockSource</i>	watchdog clock source, see wdog_clock_source_t .

26.1.3.8 static wdog_clock_source_t WDOG_HAL_GetClockSourceMode (uint32_t *baseAddr*) [inline], [static]

This function gets the WDOG clock source. There are two clock sources that can be used: the LPO clock and the bus clock. A Clock Switching Delay time is about 2 clock A cycles plus 2 clock B, where clock A and B are the two input clocks to the clock mux.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

watchdog clock source, see [wdog_clock_source_t](#).

26.1.3.9 static void WDOG_HAL_SetWindowModeCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function configures the WDOG window mode. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>enable</i>	false means disable watchdog window mode. true means enable watchdog window mode.

26.1.3.10 static bool WDOG_HAL_GetWindowModeCmd (uint32_t *baseAddr*) [inline], [static]

This function checks whether the WDOG window mode is enabled.

WDOG HAL driver

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

false means window mode is disabled, true means window mode is enabled.

26.1.3.11 static void WDOG_HAL_SetRegisterUpdateCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function configures the WDOG register update feature. If disabled, it means that all WDOG registers is never written again unless Power On Reset. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>enable</i>	false means disable watchdog write-once-only register update. true means enable watchdog write-once-only register update.

26.1.3.12 static bool WDOG_HAL_GetRegisterUpdateCmd (*uint32_t baseAddr*) [inline], [static]

This function checks whether the WDOG register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

false means register update is disabled, true means register update is enabled.

26.1.3.13 static void WDOG_HAL_SetWorkInDebugModeCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function configures whether the WDOG is enabled in the CPU debug mode. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>enable</i>	false means watchdog is disabled in CPU debug mode. true means watchdog is enabled in CPU debug mode.

26.1.3.14 static bool WDOG_HAL_GetWorkInDebugModeCmd (uint32_t *baseAddr*) [inline], [static]

This function checks whether the WDOG works in the CPU debug mode.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

false means not work while in CPU debug mode, true means works while in CPU debug mode.

26.1.3.15 static void WDOG_HAL_SetWorkInStopModeCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function configures whether the WDOG is enabled in the CPU stop mode. Make sure that the WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>enable</i>	false means watchdog is disabled in CPU stop mode. true means watchdog is enabled in CPU stop mode.

26.1.3.16 static bool WDOG_HAL_GetWorkInStopModeCmd (uint32_t *baseAddr*) [inline], [static]

This function checks whether the WDOG works in the CPU stop mode. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

WDOG HAL driver

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

false means not work while in CPU stop mode, true means works while in CPU stop mode.

26.1.3.17 static void WDOG_HAL_SetWorkInWaitModeCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function configures whether the WDOG is enabled in the CPU wait mode. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock, that the WCT window is still open and that the WDOG_STCTRLH register has not been written in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>enable</i>	false means watchdog is disabled in CPU wait mode. true means watchdog is enabled in CPU wait mode.

26.1.3.18 static bool WDOG_HAL_GetWorkInWaitModeCmd (*uint32_t baseAddr*) [inline], [static]

This function checks whether the WDOG works in the CPU wait mode.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

false means not work while in CPU wait mode, true means works while in CPU wait mode.

26.1.3.19 static bool WDOG_HAL_IsIntPending (*uint32_t baseAddr*) [inline], [static]

This function gets the WDOG interrupt flag.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

Watchdog interrupt status, false means interrupt not asserted, true means interrupt asserted.

26.1.3.20 static void WDOG_HAL_ClearIntFlag (*uint32_t baseAddr*) [inline], [static]

This function clears the WDOG interrupt flag.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.21 static void WDOG_HAL_SetTimeoutValue (*uint32_t baseAddr, uint32_t timeoutCount*) [inline], [static]

This function sets the WDOG_TOVAL value. It should be ensured that the time-out value for the Watchdog is always greater than 2xWCT time + 20 bus clock cycles. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock , that the WCT window is still open and that this API has not been called in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>timeoutCount</i>	watchdog timeout value, count of watchdog clock tick.

26.1.3.22 static uint32_t WDOG_HAL_GetTimeoutValue (*uint32_t baseAddr*) [inline], [static]

This function gets the WDOG_TOVAL value.

Parameters

WDOG HAL driver

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

value of register WDOG_TOVAL.

26.1.3.23 static uint32_t WDOG_HAL_GetTimerOutputValue (*uint32_t baseAddr*) [inline], [static]

This function gets the WDOG_TMROUT value.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

Current value of watchdog timer counter.

26.1.3.24 static void WDOG_HAL_SetClockPrescalerValueMode (*uint32_t baseAddr*, *wdog_clock_prescaler_value_t clockPrescaler*) [inline], [static]

This function sets the WDOG clock prescaler. Make sure WDOG registers are unlocked by the WDO-G_HAL_Unlock , that the WCT window is still open and that this API has not been called in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>clockPrescaler</i>	watchdog clock prescaler, see wdog_clock_prescaler_value_t .

26.1.3.25 static *wdog_clock_prescaler_value_t* WDOG_HAL_GetClockPrescalerValueMode (*uint32_t baseAddr*) [inline], [static]

This function gets the WDOG clock prescaler.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

WDOG clock prescaler, see [wdog_clock_prescaler_value_t](#).

26.1.3.26 static void WDOG_HAL_SetWindowValue (*uint32_t baseAddr, uint32_t windowValue*) [inline], [static]

This function sets the WDOG_WIN value. Make sure WDOG registers are unlocked by the WDOG_HAL_Unlock , that the WCT window is still open and that this API has not been called in this WCT while this function is called. Make sure WDOG_STCTRLH.ALLOWUPDATE is 1 which means register update is enabled.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
<i>windowValue</i>	watchdog window value.

26.1.3.27 static uint32_t WDOG_HAL_GetWindowValue (*uint32_t baseAddr*) [inline], [static]

This function gets the WDOG_WIN value.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

watchdog window value.

26.1.3.28 static void WDOG_HAL_Unlock (*uint32_t baseAddr*) [inline], [static]

This function unlocks the WDOG register written. This function must be called before any configuration is set because watchdog register will be locked automatically after a WCT(256 bus cycles).

WDOG HAL driver

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.29 static void WDOG_HAL_Refresh(uint32_t *baseAddr*) [inline], [static]

This function feeds the WDOG. This function should be called before watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.30 static void WDOG_HAL_ResetSystem(uint32_t *baseAddr*) [inline], [static]

This function resets the chip using WDOG.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.31 static uint32_t WDOG_HAL_GetResetCount(uint32_t *baseAddr*) [inline], [static]

This function gets the value of the WDOG_RSTCNT.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

Returns

Chip reset count that was reset by Watchdog.

26.1.3.32 static void WDOG_HAL_ClearResetCount(uint32_t *baseAddr*) [inline], [static]

This function clears the WDOG_RSTCNT.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

26.1.3.33 void WDOG_HAL_Init(uint32_t *baseAddr*)

This function restores the WDOG module to reset value.

Parameters

<i>baseAddr</i>	The WDOG peripheral base address
-----------------	----------------------------------

WDOG Peripheral Driver

26.2 WDOG Peripheral Driver

This chapter describes the programming interface of the WDOG Peripheral driver.

Data Structures

- struct `wdog_user_config_t`
Data structure for watchdog initialize. [More...](#)

Watchdog Driver

- void `WDOG_DRV_Init` (const `wdog_user_config_t` *userConfigPtr)
Initializes the Watchdog.
- void `WDOG_DRV_Deinit` (void)
Shuts down the Watchdog.
- void `WDOG_DRV_Refresh` (void)
Refreshes the Watchdog.
- uint32_t `WDOG_DRV_GetResetCount` (void)
Gets the MCU reset count that is reset by the Watchdog.
- void `WDOG_DRV_ClearResetCount` (void)
Clears the Watchdog reset count.
- bool `WDOG_DRV_IsRunning` (void)
Gets the Watchdog running status.
- void `WDOG_DRV_ResetSystem` (void)
Resets the MCU by the Watchdog.

26.2.0.34 WDOG Peripheral Driver

Overview

The WDOG driver is used to configure the WDOG. It also provides an easy way to initialize and configure the WDOG.

Initialization

To initialize the WDOG module, call the `WDOG_DRV_Init()` function and pass in the user configuration structure. This function automatically enables the WDOG module and clock.

After the `WDOG_DRV_Init()` function is called, the WDOG is enabled and its counter is working. Therefore, the `WDOG_DRV_Refresh()` function should be called before the WDOG times out.

This example code shows how to initialize and configure the driver:

```
// Define device configuration.  
const wdog_user_config_t init =  
{
```

```

.clockSource = kWdogClockSourceLpoClock, /* WDOG clock source is LPO
    clock */
.clockPrescalerValue = kWdogClockPrescalerValueDevide1, /* Clock
    prescaler divide by 1 */
.timeoutValue = 2048, /* Timeout count is 2048 */
.interruptEnable = false, /* Interrupt configure, false means disable interrupt */
.updateRegisterEnable = true, /* Enable WDOG register update after first configure */
.workInWaitModeEnable = true, /* Enable WDOG while CPU is in Wait mode */
.workInStopModeEnable = true, /* Enable WDOG while CPU is in Stop mode */
};

// Initialize WDOG.
WDOG_DRV_Init(&init);

```

WDOG Refresh

After the WDOG is enabled, the [WDOG_DRV_Refresh\(\)](#) function should be called periodically to prevent the WDOG from timing out.

Otherwise, a reset is asserted. This is called the "Feed Dog".

WDOG Reset Count

The WDOG can record the reset count caused by the WDOG timeout.

1. [WDOG_DRV_GetResetCount\(\)](#) gets the reset count caused by the WDOG timeout.
2. [WDOG_DRV_ClearResetCount\(\)](#) clears the reset count caused by the WDOG timeout.

WDOG Reset System

The WDOG can be used to reset the MCU.

1. [WDOG_DRV_ResetSystem\(\)](#) is used to reset the MCU whether the WDOG is enabled or not.

WDOG interrupt

If the WDOG interrupt is enabled, the WDOG asserts an interrupt and resets the system after 256 bus clock.

1. Enable the WDOG interrupt with the `wdog_user_config_t.interruptEnable = true`
2. Define the WDOG IRQ function

```

void Watchdog_IRQHandler()
{
    /* Enter WDOG ISR */
}

```

26.2.1 Data Structure Documentation

26.2.1.1 struct wdog_user_config_t

This structure is used when initializing the WDOG while the wdog_init function is called. It contains all WDOG configurations.

Data Fields

- `wdog_clock_source_t clockSource`
Clock source select.
- `wdog_clock_prescaler_value_t clockPrescalerValue`
Clock prescaler value.
- `bool updateRegisterEnable`
Update write-once register enable.
- `bool workInDebugModeEnable`
Enable watchdog while in debug mode.
- `bool workInWaitModeEnable`
Enable watchdog while in wait mode.
- `bool workInStopModeEnable`
Enable watchdog while in stop mode.
- `uint32_t windowValue`
Window value.
- `uint32_t timeoutValue`
Timeout value.

26.2.2 Function Documentation

26.2.2.1 void WDOG_DRV_Init (const wdog_user_config_t * userConfigPtr)

This function initializes the WDOG. When called, the WDOG runs according to the configuration.

Parameters

<code>userConfigPtr</code>	Watchdog user configure data structure, see wdog_user_config_t .
----------------------------	--

26.2.2.2 void WDOG_DRV_Deinit (void)

This function shuts down the WDOG.

26.2.2.3 void WDOG_DRV_Refresh (void)

This function feeds the WDOG. It sets the WDOG timer count to zero and should be called before the Watchdog timer times out. Otherwise, a reset is asserted. Enough time should be allowed for the refresh

sequence to be detected by the Watchdog timer on the Watchdog clock.

26.2.2.4 `uint32_t WDOG_DRV_GetResetCount(void)`

This function gets the WDOG_RSTCNT value.

Returns

Chip reset count that is reset by the Watchdog.

26.2.2.5 `void WDOG_DRV_ClearResetCount(void)`

This function sets the WDOG reset count to zero. The WDOG_RSTCNT register clears either on Power-On-Reset or is cleared by this function.

26.2.2.6 `bool WDOG_DRV_IsRunning(void)`

This function gets the WDOG running status.

Returns

watchdog running status, false means not running, true means running

26.2.2.7 `void WDOG_DRV_ResetSystem(void)`

This function is used to reset the MCU by using the WDOG.

Chapter 27

Low-Leakage Wakeup Unit (LLWU)

The Kinetis SDK provides a HAL driver for the Low-Leakage Wakeup Unit (LLWU) block of Kinetis devices.

Data Structures

- struct [llwu_external_pin_filter_mode_t](#)
External input pin filter control structure. [More...](#)
- struct [llwu_reset_enable_mode_t](#)
Reset enable control structure. [More...](#)

Enumerations

- enum [llwu_external_pin_modes_t](#)
External input pin control modes.
- enum [llwu_filter_modes_t](#)
Digital filter control modes.

Functions

- void [LLWU_HAL_SetExternalInputModuleMode](#) (uint32_t baseAddr, [llwu_external_pin_modes_t](#) pinMode, uint32_t pinNumber)
Sets the external input pin source mode.
- [llwu_external_pin_modes_t LLWU_HAL_GetExternalInputModuleMode](#) (uint32_t baseAddr, uint32_t pinNumber)
Gets the external input pin source mode.
- void [LLWU_HAL_SetInternalModuleCmd](#) (uint32_t baseAddr, uint32_t moduleNumber, bool enable)
Enables/disables the internal module source.
- bool [LLWU_HAL_GetInternalModuleCmd](#) (uint32_t baseAddr, uint32_t moduleNumber)
Gets the internal module source enable setting.
- bool [LLWU_HAL_GetExternalPinWakeUpFlag](#) (uint32_t baseAddr, uint32_t pinNumber)
Gets the external wake up source flag.
- void [LLWU_HAL_ClearExternalPinWakeUpFlag](#) (uint32_t baseAddr, uint32_t pinNumber)
Clears the external wake up source flag.
- bool [LLWU_HAL_GetInternalModuleWakeUpFlag](#) (uint32_t baseAddr, uint32_t moduleNumber)
Gets the internal module wake up source flag.
- void [LLWU_HAL_SetPinFilterMode](#) (uint32_t baseAddr, uint32_t filterNumber, [llwu_external_pin_filter_mode_t](#) pinFilterMode)
Sets the pin filter configuration.
- void [LLWU_HAL_GetPinFilterMode](#) (uint32_t baseAddr, uint32_t filterNumber, [llwu_external_pin_filter_mode_t](#) *pinFilterMode)
Gets the pin filter configuration.
- bool [LLWU_HAL_GetFilterDetectFlag](#) (uint32_t baseAddr, uint32_t filterNumber)

Function Documentation

- Gets the filter detect flag.
• void [LLWU_HAL_ClearFilterDetectFlag](#) (uint32_t baseAddr, uint32_t filterNumber)
Clears the filter detect flag.

27.0.3 LLWU HAL driver

Overview

The LLWU module allows the user to select up to 16 external pin sources and up to 8 internal modules as a wake-up source from low-leakage power modes.

The input sources are described in the device's chip configuration details. Each of the available wake-up sources can be individually enabled.

This is an example of LLWU HAL access APIs

```
#include "llwu/hal/fsl_llwu_hal.h"

/* set to enable pin 2 as wakeup source using rising edge */
LLWU_HAL_SetExternalInputPinMode(kLlwuExternalPinRisingEdge, 2);

/* set to enable pin 8 as wakeup source using change detected */
LLWU_HAL_SetExternalInputPinMode(kLlwuExternalPinChangeDetect, 8);

/* Set to enable internal module 1 as wakeup source */
LLWU_HAL_SetInternalModuleCmd(1, true);
```

27.1 Data Structure Documentation

27.1.1 struct llwu_external_pin_filter_mode_t

27.1.2 struct llwu_reset_enable_mode_t

27.2 Function Documentation

27.2.1 void [LLWU_HAL_SetExternalInputPinMode](#) (uint32_t *baseAddr*, llwu_external_pin_modes_t *pinMode*, uint32_t *pinNumber*)

This function sets the external input pin source mode that is used as a wake up source.

Parameters

<i>baseAddr</i>	Register base address of LLWU
-----------------	-------------------------------

<i>pinMode</i>	pin configuration mode defined in llwu_external_pin_modes_t
<i>pinNumber</i>	pin number specified

27.2.2 **llwu_external_pin_modes_t LLWU_HAL_GetExternalInputPinMode (uint32_t baseAddr, uint32_t pinNumber)**

This function gets the external input pin source mode that is used as wake up source.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>pinNumber</i>	pin number specified

Returns

pinMode pin mode defined in llwu_external_pin_modes_t

27.2.3 **void LLWU_HAL_SetInternalModuleCmd (uint32_t baseAddr, uint32_t moduleNumber, bool enable)**

This function enables/disables the internal module source mode that is used as a wake up source.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>moduleNumber</i>	module number specified
<i>enable</i>	enable or disable setting

27.2.4 **bool LLWU_HAL_GetInternalModuleCmd (uint32_t baseAddr, uint32_t moduleNumber)**

This function gets the internal module source enable setting that is used as a wake up source.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>moduleNumber</i>	module number specified

Function Documentation

Returns

enable enable or disable setting

27.2.5 bool LLWU_HAL_GetExternalPinWakeupFlag (uint32_t *baseAddr*, uint32_t *pinNumber*)

This function gets the external wakeup source flag for a specific pin.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>pinNumber</i>	pin number specified

Returns

flag true if wakeup source flag set

27.2.6 void LLWU_HAL_ClearExternalPinWakeupFlag (uint32_t *baseAddr*, uint32_t *pinNumber*)

This function clears the external wakeup source flag for a specific pin.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>pinNumber</i>	pin number specified

27.2.7 bool LLWU_HAL_GetInternalModuleWakeupFlag (uint32_t *baseAddr*, uint32_t *moduleNumber*)

This function gets the internal module wakeup source flag for a specific module.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>moduleNumber</i>	module number specified

Returns

flag true if wakeup flag set

27.2.8 void LLWU_HAL_SetPinFilterMode (*uint32_t baseAddr, uint32_t filterNumber, llwu_external_pin_filter_mode_t pinFilterMode*)

This function sets the pin filter configuration.

Function Documentation

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>filterNumber</i>	filter number specified
<i>pinFilterMode</i>	filter mode configuration

27.2.9 void LLWU_HAL_GetPinFilterMode (uint32_t *baseAddr*, uint32_t *filterNumber*, llwu_external_pin_filter_mode_t * *pinFilterMode*)

This function gets the pin filter configuration.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>filterNumber</i>	filter number specified
<i>pinFilterMode</i>	filter mode configuration

27.2.10 bool LLWU_HAL_GetFilterDetectFlag (uint32_t *baseAddr*, uint32_t *filterNumber*)

This function will get the filter detect flag.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>filterNumber</i>	filter number specified

Returns

flag true if the filter was a wakeup source

27.2.11 void LLWU_HAL_ClearFilterDetectFlag (uint32_t *baseAddr*, uint32_t *filterNumber*)

This function will clear the filter detect flag.

Parameters

<i>baseAddr</i>	Register base address of LLWU
<i>filterNumber</i>	filter number specified

Function Documentation

Chapter 28

Multipurpose Clock Generator (MCG)

The Kinetis SDK provides a HAL driver for the Multipurpose Clock Generator (MCG) block of Kinetis devices.

Modules

- [MCG HAL driver](#)

The part describes the programming interface of the MCG Hal driver.

28.1 MCG HAL driver

The chapter describes the programming interface of the MCG Hal driver.

Enumerations

- enum `_mcg_constant`
MCG constant definitions.
- enum `mcg_clock_select_t`
MCG clock source select.
- enum `mcg_internal_ref_clock_source_t`
MCG internal reference clock source select.
- enum `mcg_freq_range_select_t`
MCG frequency range select.
- enum `mcg_high_gain_osc_select_t`
MCG high gain oscillator select.
- enum `mcg_external_ref_clock_select_t`
MCG high gain oscillator select.
- enum `mcg_low_power_select_t`
MCG low power select.
- enum `mcg_internal_ref_clock_select_t`
MCG internal reference clock select.
- enum `mcg_dmx32_select_t`
MCG DCO Maximum Frequency with 32.768 kHz Reference.
- enum `mcg_digital_controlled_osc_range_select_t`
MCG DCO range select.
- enum `mcg_pll_external_ref_clk_select_t`
MCG PLL external reference clock select.
- enum `mcg_pll_select_t`
MCG PLL select.
- enum `mcg_loss_of_lock_status_t`
MCG loss of lock status.
- enum `mcg_lock_status_t`
MCG lock status.
- enum `mcg_pll_stat_status_t`
MCG clock status.
- enum `mcg_internal_ref_status_t`
MCG iref status.
- enum `mcg_clk_stat_status_t`
MCG clock mode status.
- enum `mcg_internal_ref_clk_status_t`
MCG ircst status.
- enum `mcg_auto_trim_machine_fail_status_t`
MCG auto trim fail status.
- enum `mcg_locs0_status_t`
MCG loss of clock status.
- enum `mcg_auto_trim_machine_select_t`
MCG Automatic Trim Machine Select.
- enum `mcg_oscsel_select_t`
MCG OSC Clock Select.
- enum `mcg_loss_of_clk1_status_t`

- *MCG loss of clock status.*
- enum `mcg_pll_clk_select_t`
MCG PLLCS select.
- enum `mcg_locs2_status_t`
MCG loss of clock status.
- enum `mcg_modes_t`
MCG mode definitions.
- enum `mcg_mode_error_code_t`
MCG mode transition API error code definitions.

Functions

- `mcg_modes_t CLOCK_HAL_GetMcgMode (uint32_t baseAddr)`
Gets the current MCG mode.
- `uint32_t CLOCK_HAL_GetFllFrequency (uint32_t baseAddr, int32_t fllRef)`
Checks the FLL frequency integrity.
- `uint32_t CLOCK_HAL_SetFeiToFeeMode (uint32_t baseAddr, mcg_oscsel_select_t oscselVal, uint32_t crystalVal, mcg_high_gain_osc_select_t hgoVal, mcg_external_ref_clock_select_t erefsVal)`
Mode transition FEI to FEE mode.
- `uint32_t CLOCK_HAL_SetFeiToFbiMode (uint32_t baseAddr, uint32_t ircFreq, mcg_internal_ref_clock_select_t ircSelect)`
Mode transition FEI to FBI mode.
- `uint32_t CLOCK_HAL_SetFeiToFbeMode (uint32_t baseAddr, mcg_oscsel_select_t oscselVal, uint32_t crystalVal, mcg_high_gain_osc_select_t hgoVal, mcg_external_ref_clock_select_t erefsVal)`
Mode transition FEI to FBE mode.
- `uint32_t CLOCK_HAL_SetFeeToFeiMode (uint32_t baseAddr, uint32_t ircFreq)`
Mode transition FEE to FEI mode.
- `uint32_t CLOCK_HAL_SetFeeToFbiMode (uint32_t baseAddr, uint32_t ircFreq, mcg_internal_ref_clock_select_t ircSelect)`
Mode transition FEE to FBI mode.
- `uint32_t CLOCK_HAL_SetFeeToFbeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition FEE to FBE mode.
- `uint32_t CLOCK_HAL_SetFbiToFeiMode (uint32_t baseAddr, uint32_t ircFreq)`
Mode transition FBI to FEI mode.
- `uint32_t CLOCK_HAL_SetFbiToFeeMode (uint32_t baseAddr, mcg_oscsel_select_t oscselVal, uint32_t crystalVal, mcg_high_gain_osc_select_t hgoVal, mcg_external_ref_clock_select_t erefsVal)`
Mode transition FBI to FEE mode.
- `uint32_t CLOCK_HAL_SetFbiToFbeMode (uint32_t baseAddr, mcg_oscsel_select_t oscselVal, uint32_t crystalVal, mcg_high_gain_osc_select_t hgoVal, mcg_external_ref_clock_select_t erefsVal)`
Mode transition FBI to FBE mode.
- `uint32_t CLOCK_HAL_SetFbiToBlpiMode (uint32_t baseAddr, uint32_t ircFreq, mcg_internal_ref_clock_select_t ircSelect)`
Mode transition FBI to BLPI mode.
- `uint32_t CLOCK_HAL_SetBlpiToFbiMode (uint32_t baseAddr, uint32_t ircFreq, uint8_t ircSelect)`

MCG HAL driver

- `uint32_t CLOCK_HAL_SetFbeToFeeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition BLPI to FBI mode.
- `uint32_t CLOCK_HAL_SetFbeToFeiMode (uint32_t baseAddr, uint32_t ircFreq)`
Mode transition FBE to FEI mode.
- `uint32_t CLOCK_HAL_SetFbeToFbiMode (uint32_t baseAddr, uint32_t ircFreq, mcg_internal_ref_clock_select_t ircSelect)`
Mode transition FBE to FBI mode.
- `uint32_t CLOCK_HAL_SetFbeToPbeMode (uint32_t baseAddr, uint32_t crystalVal, mcg_pll_clk_select_t pllcsSelect, uint8_t prdivVal, uint8_t vdivVal)`
Mode transition FBE to PBE mode.
- `uint32_t CLOCK_HAL_SetFbeToBlpeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition FBE to BLPE mode.
- `uint32_t CLOCK_HAL_SetPbeToFbeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition PBE to FBE mode.
- `uint32_t CLOCK_HAL_SetPbeToPeeMode (uint32_t baseAddr, uint32_t crystalVal, mcg_pll_clk_select_t pllcsSelect)`
Mode transition PBE to PEE mode.
- `uint32_t CLOCK_HAL_SetPbeToBlpeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition PBE to BLPE mode.
- `uint32_t CLOCK_HAL_SetPeeToPbeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition PEE to PBE mode.
- `uint32_t CLOCK_HAL_SetBlpeToPbeMode (uint32_t baseAddr, uint32_t crystalVal, mcg_pll_clk_select_t pllcsSelect, uint8_t prdivVal, uint8_t vdivVal)`
Mode transition BLPE to PBE mode.
- `uint32_t CLOCK_HAL_SetBlpeToFbeMode (uint32_t baseAddr, uint32_t crystalVal)`
Mode transition BLPE to FBE mode.

MCG out clock access API

- `uint32_t CLOCK_HAL_GetFllRefClk (uint32_t baseAddr)`
Gets the current MCG FLL clock.
- `uint32_t CLOCK_HAL_GetFllClk (uint32_t baseAddr)`
Gets the current MCG FLL clock.
- `uint32_t CLOCK_HAL_GetPll0Clk (uint32_t baseAddr)`
Gets the current MCG PLL/PLL0 clock.
- `uint32_t CLOCK_HAL_GetInternalRefClk (uint32_t baseAddr)`
Gets the current MCG IR clock.
- `uint32_t CLOCK_HAL_GetOutClk (uint32_t baseAddr)`
Gets the current MCG out clock.

MCG control register access API

- static void `CLOCK_HAL_SetClkSrcMode (uint32_t baseAddr, mcg_clock_select_t select)`
Sets the Clock Source Select.
- static `mcg_clock_select_t CLOCK_HAL_GetClkSrcMode (uint32_t baseAddr)`
Gets the Clock Source Select.
- static void `CLOCK_HAL_SetFllExternalRefDivider (uint32_t baseAddr, uint8_t setting)`

- static uint8_t **CLOCK_HAL_GetFLLExternalRefDivider** (uint32_t baseAddr)

Gets the FLL External Reference Divider.
- static void **CLOCK_HAL_SetInternalRefSelMode** (uint32_t baseAddr, **mcg_internal_ref_clock_source_t** select)

Sets the Internal Reference Select.
- static **mcg_internal_ref_clock_source_t CLOCK_HAL_GetInternalRefSelMode** (uint32_t baseAddr)

Gets the Internal Reference Select.
- static void **CLOCK_HAL_SetClksFrdivInternalRefSelect** (uint32_t baseAddr, **mcg_clock_select_t** clks, uint8_t frdiv, **mcg_internal_ref_clock_source_t** irefs)

Sets the CLKS, FRDIV and IREFS at the same time.
- static void **CLOCK_HAL_SetInternalClkCmd** (uint32_t baseAddr, bool enable)

Sets the Enable Internal Reference Clock setting.
- static bool **CLOCK_HAL_GetInternalClkCmd** (uint32_t baseAddr)

Gets the enable Internal Reference Clock setting.
- static void **CLOCK_HAL_SetInternalRefStopCmd** (uint32_t baseAddr, bool enable)

Sets the Internal Reference Clock Stop Enable setting.
- static bool **CLOCK_HAL_GetInternalRefStopCmd** (uint32_t baseAddr)

Gets the Enable Internal Reference Clock setting.
- static void **CLOCK_HAL_SetLossOfClkReset0Cmd** (uint32_t baseAddr, bool enable)

Sets the Loss of Clock Reset Enable setting.
- static bool **CLOCK_HAL_GetLossOfClkReset0Cmd** (uint32_t baseAddr)

Gets the Loss of Clock Reset Enable setting.
- static void **CLOCK_HAL_SetRange0Mode** (uint32_t baseAddr, **mcg_freq_range_select_t** select)

Sets the Frequency Range Select.
- static **mcg_freq_range_select_t CLOCK_HAL_GetRange0Mode** (uint32_t baseAddr)

Gets the Frequency Range Select.
- static void **CLOCK_HAL_SetHighGainOsc0Mode** (uint32_t baseAddr, **mcg_high_gain_osc_select_t** select)

Sets the High Gain Oscillator Select.
- static **mcg_high_gain_osc_select_t CLOCK_HAL_GetHighGainOsc0Mode** (uint32_t baseAddr)

Gets the High Gain Oscillator Select.
- static void **CLOCK_HAL_SetExternalRefSel0Mode** (uint32_t baseAddr, **mcg_external_ref_clock_select_t** select)

Sets the External Reference Select.
- static **mcg_external_ref_clock_select_t CLOCK_HAL_GetExternalRefSel0Mode** (uint32_t baseAddr)

Gets the External Reference Select.
- static void **CLOCK_HAL_SetLowPowerMode** (uint32_t baseAddr, **mcg_low_power_select_t** select)

Sets the Low Power Select.
- static **mcg_low_power_select_t CLOCK_HAL_GetLowPowerMode** (uint32_t baseAddr)

Gets the Low Power Select.
- static void **CLOCK_HAL_SetInternalRefClkSelMode** (uint32_t baseAddr, **mcg_internal_ref_clock_select_t** select)

Sets the Internal Reference Clock Select.
- static **mcg_internal_ref_clock_select_t CLOCK_HAL_GetInternalRefClkSelMode** (uint32_t baseAddr)

Gets the Internal Reference Clock Select.

MCG HAL driver

- static void **CLOCK_HAL_SetSlowInternalRefClkTrim** (uint32_t baseAddr, uint8_t setting)
Sets the Slow Internal Reference Clock Trim Setting.
- static uint8_t **CLOCK_HAL_GetSlowInternalRefClkTrim** (uint32_t baseAddr)
Gets the Slow Internal Reference Clock Trim Setting.
- static void **CLOCK_HAL_SetDmx32** (uint32_t baseAddr, **mcg_dmx32_select_t** setting)
Sets the DCO Maximum Frequency with 32.768 kHz Reference.
- static **mcg_dmx32_select_t CLOCK_HAL_GetDmx32** (uint32_t baseAddr)
Gets the DCO Maximum Frequency with the 32.768 kHz Reference Setting.
- static void **CLOCK_HAL_SetDigitalControlledOscRangeMode** (uint32_t baseAddr, **mcg_digital_controlled_osc_range_select_t** setting)
Sets the DCO Range Select.
- static **mcg_digital_controlled_osc_range_select_t CLOCK_HAL_GetDigitalControlledOscRangeMode** (uint32_t baseAddr)
Gets the DCO Range Select Setting.
- static void **CLOCK_HAL_SetFastInternalRefClkTrim** (uint32_t baseAddr, uint8_t setting)
Sets the Fast Internal Reference Clock Trim Setting.
- static uint8_t **CLOCK_HAL_GetFastInternalRefClkTrim** (uint32_t baseAddr)
Gets the Fast Internal Reference Clock Trim Setting.
- static void **CLOCK_HAL_SetSlowInternalRefClkFineTrim** (uint32_t baseAddr, uint8_t setting)
Sets the Slow Internal Reference Clock Fine Trim Setting.
- static uint8_t **CLOCK_HAL_GetSlowInternalRefClkFineTrim** (uint32_t baseAddr)
Gets the Slow Internal Reference Clock Fine Trim Setting.
- static **mcg_internal_ref_status_t CLOCK_HAL_GetInternalRefStatMode** (uint32_t baseAddr)
Gets the Internal Reference Status.
- static **mcg_clk_stat_status_t CLOCK_HAL_GetClkStatMode** (uint32_t baseAddr)
Gets the Clock Mode Status.
- static uint8_t **CLOCK_HAL_GetOscInit0** (uint32_t baseAddr)
Gets the OSC Initialization Status.
- static **mcg_internal_ref_clk_status_t CLOCK_HAL_GetInternalRefClkStatMode** (uint32_t baseAddr)
Gets the Internal Reference Clock Status.
- static **mcg_auto_trim_machine_fail_status_t CLOCK_HAL_GetAutoTrimMachineFailMode** (uint32_t baseAddr)
Gets the Automatic Trim machine Fail Flag.
- static void **CLOCK_HAL_SetAutoTrimMachineFail** (uint32_t baseAddr)
Sets the Automatic Trim machine Fail Flag.
- static **mcg_locs0_status_t CLOCK_HAL_GetLocs0Mode** (uint32_t baseAddr)
Gets the OSC0 Loss of Clock Status.
- static void **CLOCK_HAL_SetAutoTrimMachineCmd** (uint32_t baseAddr, bool enable)
Sets the Automatic Trim Machine Enable Setting.
- static bool **CLOCK_HAL_GetAutoTrimMachineCmd** (uint32_t baseAddr)
Gets the Automatic Trim Machine Enable Setting.
- static void **CLOCK_HAL_SetAutoTrimMachineSelMode** (uint32_t baseAddr, **mcg_auto_trim_machine_select_t** setting)
Sets the Automatic Trim Machine Select Setting.
- static **mcg_auto_trim_machine_select_t CLOCK_HAL_GetAutoTrimMachineSelMode** (uint32_t baseAddr)
Gets the Automatic Trim Machine Select Setting.

- static void **CLOCK_HAL_SetFllFilterPreserveCmd** (uint32_t baseAddr, bool enable)

Gets the Automatic Trim Machine Select Setting.

Sets the FLL Filter Preserve Enable Setting.
- static bool **CLOCK_HAL_GetFllFilterPreserveCmd** (uint32_t baseAddr)

Gets the FLL Filter Preserve Enable Setting.
- static void **CLOCK_HAL_SetFastClkInternalRefDivider** (uint32_t baseAddr, uint8_t setting)

Gets the Fast Clock Internal Reference Divider Setting.
- static uint8_t **CLOCK_HAL_GetFastClkInternalRefDivider** (uint32_t baseAddr)

Gets the Fast Clock Internal Reference Divider Setting.
- static void **CLOCK_HAL_SetAutoTrimMachineCompValHigh** (uint32_t baseAddr, uint8_t setting)

Sets the ATM Compare Value High Setting.
- static uint8_t **CLOCK_HAL_GetAutoTrimMachineCompValHigh** (uint32_t baseAddr)

Gets the ATM Compare Value High Setting.
- static void **CLOCK_HAL_SetAutoTrimMachineCompValLow** (uint32_t baseAddr, uint8_t setting)

Sets the ATM Compare Value Low Setting.
- static uint8_t **CLOCK_HAL_GetAutoTrimMachineCompValLow** (uint32_t baseAddr)

Gets the ATM Compare Value Low Setting.

28.1.0.1 MCG Hal Driver

Overview

The multipurpose clock generator (MCG) module provides a several clock source choices for the MCU. The MCG HAL provides a set of APIs to accessing these registers.

Get current default reference clock frequency

The MCG HAL provides a set of APIs dedicated to get the system default clock frequency. For example, MCG out clock, FLL clock, PLL clock, etc.

Example to get a default system reference clock:

```
#include "mcg/hal/fsl_mcg_hal.h"

// return frequency value for specified clock name
uint32_t frequency = 0;

// get the current system default reference clock frequency
frequency = CLOCK_HAL_GetOutclk();
```

28.1.1 Function Documentation

28.1.1.1 uint32_t **CLOCK_HAL_GetFllRefClk** (uint32_t *baseAddr*)

This function returns the mcgfllclk value in frequency(Hertz) based on the current MCG configurations and settings. FLL should be properly configured in order to get the valid value.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

value Frequency value in Hertz of the mcgpllclk.

28.1.1.2 `uint32_t CLOCK_HAL_GetFllClk (uint32_t baseAddr)`

This function returns the mcgflclk value in frequency(Hertz) based on the current MCG configurations and settings. FLL should be properly configured in order to get the valid value.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

value Frequency value in Hertz of the mcgpllclk.

28.1.1.3 `uint32_t CLOCK_HAL_GetPll0Clk (uint32_t baseAddr)`

This function returns the mcgpllclk/mcgpll0 value in frequency(Hertz) based on the current MCG configurations and settings. PLL/PLL0 should be properly configured in order to get the valid value.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

value Frequency value in Hertz of the mcgpllclk or the mcgpll0clk.

28.1.1.4 `uint32_t CLOCK_HAL_GetInternalRefClk (uint32_t baseAddr)`

This function returns the mcgirclk value in frequency (Hertz) based on the current MCG configurations and settings. It does not check if the mcgirclk is enabled or not, just calculate and return the value.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

value Frequency value in Hertz of the mcgirclk.

28.1.1.5 **uint32_t CLOCK_HAL_GetOutClk (uint32_t *baseAddr*)**

This function returns the mcgoutclk value in frequency (Hertz) based on the current MCG configurations and settings. The configuration should be properly done in order to get the valid value.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

value Frequency value in Hertz of mcgoutclk.

28.1.1.6 **static void CLOCK_HAL_SetClkSrcMode (uint32_t *baseAddr*, mcg_clock_select_t *select*) [inline], [static]**

This function selects the clock source for the MCGOUTCLK.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>select</i>	Clock source selection <ul style="list-style-type: none"> • 00: Output of FLL or PLLCS is selected(depends on PLLS control bit) • 01: Internal reference clock is selected. • 10: External reference clock is selected. • 11: Reserved.

28.1.1.7 **static mcg_clock_select_t CLOCK_HAL_GetClkSrcMode (uint32_t *baseAddr*) [inline], [static]**

This function gets the select of the clock source for the MCGOUTCLK.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select Clock source selection

28.1.1.8 static void CLOCK_HAL_SetFLLExternalRefDivider (*uint32_t baseAddr, uint8_t setting*) [inline], [static]

This function sets the FLL External Reference Divider.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>setting</i>	Divider setting

28.1.1.9 static uint8_t CLOCK_HAL_GetFLLExternalRefDivider (*uint32_t baseAddr*) [inline], [static]

This function gets the FLL External Reference Divider.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting Divider setting

28.1.1.10 static void CLOCK_HAL_SetInternalRefSelMode (*uint32_t baseAddr, mcg_internal_ref_clock_source_t select*) [inline], [static]

This function selects the reference clock source for the FLL.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>select</i>	Clock source select <ul style="list-style-type: none"> • 0: External reference clock is selected • 1: The slow internal reference clock is selected

28.1.1.11 static mcg_internal_ref_clock_source_t CLOCK_HAL_GetInternalRefSelMode (uint32_t *baseAddr*) [inline], [static]

This function gets the reference clock source for the FLL.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select Clock source select

28.1.1.12 static void CLOCK_HAL_SetClksFrdivInternalRefSelect (uint32_t *baseAddr*, mcg_clock_select_t *clks*, uint8_t *frdiv*, mcg_internal_ref_clock_source_t *irefs*) [inline], [static]

This function sets the CLKS, FRDIV, and IREFS settings at the same time in order keep the integrity of the clock switching.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>clks</i>	Clock source select
<i>frdiv</i>	FLL external reference divider select
<i>irefs</i>	Internal reference select

28.1.1.13 static void CLOCK_HAL_SetInternalClkCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables/disables the internal reference clock to use as the MCGIRCLK.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance. enable Enable or disable internal reference clock. <ul style="list-style-type: none">• true: MCGIRCLK active• false: MCGIRCLK inactive
-----------------	--

28.1.1.14 static bool CLOCK_HAL_GetInternalClkCmd (uint32_t *baseAddr*) [inline], [static]

This function gets the reference clock enable setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

enabled True if the internal reference clock is enabled.

28.1.1.15 static void CLOCK_HAL_SetInternalRefStopCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function controls whether or not the internal reference clock remains enabled when the MCG enters Stop mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. enable Enable or disable the internal reference clock stop setting. <ul style="list-style-type: none">• true: Internal reference clock is enabled in Stop mode if IRCLKEN is set or if MCG is in FEI, FBI, or BLPI modes before entering Stop mode.• false: Internal reference clock is disabled in Stop mode
-----------------	---

28.1.1.16 static bool CLOCK_HAL_GetInternalRefStopCmd (uint32_t *baseAddr*) [inline], [static]

This function gets the Internal Reference Clock Stop Enable setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

enabled True if internal reference clock stop is enabled.

28.1.1.17 static void CLOCK_HAL_SetLossOfClkReset0Cmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function determines whether an interrupt or a reset request is made following a loss of the OSC0 external reference clock. The LOCRE0 only has an affect when CME0 is set.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. enable Loss of Clock Reset Enable setting <ul style="list-style-type: none"> • true: Generate a reset request on a loss of OSC0 external reference clock • false: Interrupt request is generated on a loss of OSC0 external reference clock
-----------------	--

28.1.1.18 static bool CLOCK_HAL_GetLossOfClkReset0Cmd (uint32_t *baseAddr*) [inline], [static]

This function gets the Loss of Clock Reset Enable setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

enabled True if Loss of Clock Reset is enabled.

28.1.1.19 static void CLOCK_HAL_SetRange0Mode (uint32_t *baseAddr*, mcg_freq_range_select_t *select*) [inline], [static]

This function selects the frequency range for the crystal oscillator or an external clock source. See the Oscillator (OSC) chapter for more details and the device data sheet for the frequency ranges used.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance. select Frequency Range Select <ul style="list-style-type: none">• 00: Low frequency range selected for the crystal oscillator• 01: High frequency range selected for the crystal oscillator• 1X: Very high frequency range selected for the crystal oscillator
-----------------	---

28.1.1.20 static mcg_freq_range_select_t CLOCK_HAL_GetRange0Mode (uint32_t baseAddr) [inline], [static]

This function gets the Frequency Range Select.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select Frequency Range Select

28.1.1.21 static void CLOCK_HAL_SetHighGainOsc0Mode (uint32_t baseAddr, mcg_high_gain_osc_select_t select) [inline], [static]

This function controls the crystal oscillator mode of operation. See the Oscillator (OSC) chapter for more details.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. select High Gain Oscillator Select. <ul style="list-style-type: none">• 0: Configure crystal oscillator for low-power operation• 1: Configure crystal oscillator for high-gain operation
-----------------	--

28.1.1.22 static mcg_high_gain_osc_select_t CLOCK_HAL_GetHighGainOsc0Mode (uint32_t baseAddr) [inline], [static]

This function gets the High Gain Oscillator Select.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select High Gain Oscillator Select

28.1.1.23 static void CLOCK_HAL_SetExternalRefSel0Mode (uint32_t *baseAddr*, mcg_external_ref_clock_select_t *select*) [inline], [static]

This function selects the source for the external reference clock. See the Oscillator (OSC) chapter for more details.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. select External Reference Select <ul style="list-style-type: none"> • 0: External reference clock requested • 1: Oscillator requested
-----------------	--

28.1.1.24 static mcg_external_ref_clock_select_t CLOCK_HAL_GetExternalRefSel0Mode (uint32_t *baseAddr*) [inline], [static]

This function gets the External Reference Select.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select External Reference Select

28.1.1.25 static void CLOCK_HAL_SetLowPowerMode (uint32_t *baseAddr*, mcg_low_power_select_t *select*) [inline], [static]

This function controls whether the FLL (or PLL) is disabled in the BLPI and the BLPE modes. In the FBE or the PBE modes, setting this bit to 1 transitions the MCG into the BLPE mode; in the FBI mode, setting this bit to 1 transitions the MCG into the BLPI mode. In any other MCG mode, the LP bit has no affect..

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance. select Low Power Select <ul style="list-style-type: none">• 0: FLL (or PLL) is not disabled in bypass modes• 1: FLL (or PLL) is disabled in bypass modes (lower power)
-----------------	--

28.1.1.26 static mcg_low_power_select_t CLOCK_HAL_GetLowPowerMode (uint32_t *baseAddr*) [inline], [static]

This function gets the Low Power Select.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select Low Power Select

28.1.1.27 static void CLOCK_HAL_SetInternalRefClkSelMode (uint32_t *baseAddr*, mcg_internal_ref_clock_select_t *select*) [inline], [static]

This function selects between the fast or slow internal reference clock source.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. select Low Power Select <ul style="list-style-type: none">• 0: Slow internal reference clock selected.• 1: Fast internal reference clock selected.
-----------------	--

28.1.1.28 static mcg_internal_ref_clock_select_t CLOCK_HAL_GetInternalRefClkSelMode (uint32_t *baseAddr*) [inline], [static]

This function gets the Internal Reference Clock Select.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

select Internal Reference Clock Select

28.1.1.29 static void CLOCK_HAL_SetSlowInternalRefClkTrim (*uint32_t baseAddr, uint8_t setting*) [inline], [static]

This function controls the slow internal reference clock frequency by controlling the slow internal reference clock period. The SCTRIM bits are binary weighted (that is, bit 1 adjusts twice as much as bit 0). Increasing the binary value increases the period, and decreasing the value decreases the period. An additional fine trim bit is available in the C4 register as the SCFTRIM bit. Upon reset, this value is loaded with a factory trim value. If an SCTRIM value stored in non-volatile memory is to be used, it is the user's responsibility to copy that value from the non-volatile memory location to this register.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting Slow Internal Reference Clock Trim Setting
-----------------	---

28.1.1.30 static uint8_t CLOCK_HAL_GetSlowInternalRefClkTrim (*uint32_t baseAddr*) [inline], [static]

This function gets the Slow Internal Reference Clock Trim Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting Slow Internal Reference Clock Trim Setting

28.1.1.31 static void CLOCK_HAL_SetDmx32 (*uint32_t baseAddr, mcg_dmx32_select_t setting*) [inline], [static]

This function controls whether or not the DCO frequency range is narrowed to its maximum frequency with a 32.768 kHz reference.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting DCO Maximum Frequency with 32.-768 kHz Reference Setting <ul style="list-style-type: none">• 0: DCO has a default range of 25%.• 1: DCO is fine-tuned for maximum frequency with 32.768 kHz reference.
-----------------	--

28.1.1.32 static mcg_dmx32_select_t CLOCK_HAL_GetDmx32 (uint32_t *baseAddr*) [inline], [static]

This function gets the DCO Maximum Frequency with 32.768 kHz Reference Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting DCO Maximum Frequency with 32.768 kHz Reference Setting

28.1.1.33 static void CLOCK_HAL_SetDigitalControlledOscRangeMode (uint32_t *baseAddr*, mcg_digital_controlled_osc_range_select_t *setting*) [inline], [static]

This function selects the frequency range for the FLL output, DCOOUT. When the LP bit is set, the writes to the DRS bits are ignored. The DRST read field indicates the current frequency range for the DCOOUT. The DRST field does not update immediately after a write to the DRS field due to internal synchronization between the clock domains. See the DCO Frequency Range table for more details.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting DCO Range Select Setting <ul style="list-style-type: none">• 00: Low range (reset default).• 01: Mid range.• 10: Mid-high range.• 11: High range.
-----------------	---

28.1.1.34 static mcg_digital_controlled_osc_range_select_t CLOCK_HAL_GetDigitalControlledOscRangeMode (uint32_t *baseAddr*) [inline], [static]

This function gets the DCO Range Select Setting.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting DCO Range Select Setting

28.1.1.35 static void CLOCK_HAL_SetFastInternalRefClkTrim (*uint32_t baseAddr*, *uint8_t setting*) [inline], [static]

This function controls the fast internal reference clock frequency by controlling the fast internal reference clock period. The FCTRIM bits are binary weighted (that is, bit 1 adjusts twice as much as bit 0). Increasing the binary value increases the period, and decreasing the value decreases the period. If an F-CTRIM[3:0] value stored in non-volatile memory is to be used, it is the user's responsibility to copy that value from the non-volatile memory location to this register.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting Fast Internal Reference Clock Trim Setting.
-----------------	--

28.1.1.36 static uint8_t CLOCK_HAL_GetFastInternalRefClkTrim (*uint32_t baseAddr*) [inline], [static]

This function gets the Fast Internal Reference Clock Trim Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting Fast Internal Reference Clock Trim Setting

28.1.1.37 static void CLOCK_HAL_SetSlowInternalRefClkFineTrim (*uint32_t baseAddr*, *uint8_t setting*) [inline], [static]

This function controls the smallest adjustment of the slow internal reference clock frequency. Setting the SCFTRIM increases the period and clearing the SCFTRIM decreases the period by the smallest amount possible. If an SCFTRIM value, stored in non-volatile memory, is to be used, it is the user's responsibility to copy that value from the non-volatile memory location to this bit.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting Slow Internal Reference Clock Fine Trim Setting
-----------------	--

28.1.1.38 static uint8_t CLOCK_HAL_GetSlowInternalRefClkFineTrim (*uint32_t baseAddr*) [inline], [static]

This function gets the Slow Internal Reference Clock Fine Trim Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting Slow Internal Reference Clock Fine Trim Setting

28.1.1.39 static mcg_internal_ref_status_t CLOCK_HAL_GetInternalRefStatMode (*uint32_t baseAddr*) [inline], [static]

This function gets the Internal Reference Status. This bit indicates the current source for the FLL reference clock. The IREFST bit does not update immediately after a write to the IREFS bit due to internal synchronization between the clock domains.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

status Internal Reference Status

- 0: Source of FLL reference clock is the external reference clock.
- 1: Source of FLL reference clock is the internal reference clock.

28.1.1.40 static mcg_clk_stat_status_t CLOCK_HAL_GetClkStatMode (*uint32_t baseAddr*) [inline], [static]

This function gets the Clock Mode Status. These bits indicate the current clock mode. The CLKST bits do not update immediately after a write to the CLKS bits due to internal synchronization between clock domains.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

status Clock Mode Status

- 00: Output of the FLL is selected (reset default).
- 01: Internal reference clock is selected.
- 10: External reference clock is selected.
- 11: Output of the PLL is selected.

28.1.1.41 static uint8_t CLOCK_HAL_GetOscInit0 (uint32_t *baseAddr*) [inline], [static]

This function gets the OSC Initialization Status. This bit, which resets to 0, is set to 1 after the initialization cycles of the crystal oscillator clock have completed. After being set, the bit is cleared to 0 if the OSC is subsequently disabled. See the OSC module's detailed description for more information.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

status OSC Initialization Status

28.1.1.42 static mcg_internal_ref_clk_status_t CLOCK_HAL_GetInternalRefClkStatMode (uint32_t *baseAddr*) [inline], [static]

This function gets the Internal Reference Clock Status. The IRCST bit indicates the current source for the internal reference clock select clock (IRCSCLK). The IRCST bit does not update immediately after a write to the IRCS bit due to the internal synchronization between clock domains. The IRCST bit is only updated if the internal reference clock is enabled, either by the MCG being in a mode that uses the IRC or by setting the C1[IRCLKEN] bit.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

status Internal Reference Clock Status

- 0: Source of internal reference clock is the slow clock (32 kHz IRC).
- 1: Source of internal reference clock is the fast clock (2 MHz IRC).

28.1.1.43 static mcg_auto_trim_machine_fail_status_t CLOCK_HAL_GetAutoTrimMachineFailMode (uint32_t *baseAddr*) [inline], [static]

This function gets the Automatic Trim machine Fail Flag. This Fail flag for the Automatic Trim Machine (ATM). This bit asserts when the Automatic Trim Machine is enabled (ATME=1) and a write to the C1, C3, C4, and SC registers is detected or the MCG enters into any Stop mode. A write to ATMF clears the flag.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

flag Automatic Trim machine Fail Flag
• 0: Automatic Trim Machine completed normally.
• 1: Automatic Trim Machine failed.

28.1.1.44 static void CLOCK_HAL_SetAutoTrimMachineFail (uint32_t *baseAddr*) [inline], [static]

This function clears the ATMF flag.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

28.1.1.45 static mcg_locs0_status_t CLOCK_HAL_GetLocs0Mode (uint32_t *baseAddr*) [inline], [static]

This function gets the OSC0 Loss of Clock Status. The LOCS0 indicates when a loss of OSC0 reference clock has occurred. The LOCS0 bit only has an effect when CME0 is set. This bit is cleared by writing a logic 1 to it when set.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

status OSC0 Loss of Clock Status
• 0: Loss of OSC0 has not occurred.
• 1: Loss of OSC0 has occurred.

MCG HAL driver

28.1.1.46 static void CLOCK_HAL_SetAutoTrimMachineCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables/disables the Auto Trim Machine to start automatically trimming the selected Internal Reference Clock. ATME de-asserts after the Auto Trim Machine has completed trimming all trim bits of the IRCS clock selected by the ATMS bit. Writing to C1, C3, C4, and SC registers or entering Stop mode aborts the auto trim operation and clears this bit.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. enable Automatic Trim Machine Enable Setting <ul style="list-style-type: none">• true: Auto Trim Machine enabled• false: Auto Trim Machine disabled
-----------------	---

28.1.1.47 static bool CLOCK_HAL_GetAutoTrimMachineCmd (uint32_t *baseAddr*) [inline], [static]

This function gets the Automatic Trim Machine Enable Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

enabled True if Automatic Trim Machine is enabled

28.1.1.48 static void CLOCK_HAL_SetAutoTrimMachineSelMode (uint32_t *baseAddr*, mcg_auto_trim_machine_select_t *setting*) [inline], [static]

This function selects the IRCS clock for Auto Trim Test.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting Automatic Trim Machine Select Setting <ul style="list-style-type: none">• 0: 32 kHz Internal Reference Clock selected• 1: 4 MHz Internal Reference Clock selected
-----------------	---

28.1.1.49 static mcg_auto_trim_machine_select_t CLOCK_HAL_GetAutoTrimMachineSelectMode (uint32_t *baseAddr*) [inline], [static]

This function gets the Automatic Trim Machine Select Setting.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting Automatic Trim Machine Select Setting

28.1.1.50 static void CLOCK_HAL_SetFllFilterPreserveCmd (*uint32_t baseAddr, bool enable*) [inline], [static]

This function sets the FLL Filter Preserve Enable. This bit prevents the FLL filter values from resetting allowing the FLL output frequency to remain the same during the clock mode changes where the FLL/-DCO output is still valid. (Note: This requires that the FLL reference frequency remain the same as the value prior to the new clock mode switch. Otherwise, the FLL filter and the frequency values change.)

Parameters

<i>baseAddr</i>	Base address for current MCG instance. enable FLL Filter Preserve Enable Setting <ul style="list-style-type: none">• true: FLL filter and FLL frequency retain their previous values during new clock mode change• false: FLL filter and FLL frequency will reset on changes to correct clock mode
-----------------	---

28.1.1.51 static bool CLOCK_HAL_GetFllFilterPreserveCmd (*uint32_t baseAddr*) [inline], [static]

This function gets the FLL Filter Preserve Enable Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

enabled True if FLL Filter Preserve is enabled.

28.1.1.52 static void CLOCK_HAL_SetFastClkInternalRefDivider (*uint32_t baseAddr, uint8_t setting*) [inline], [static]

This function selects the amount to divide down the fast internal reference clock. The resulting frequency is in the range 31.25 kHz to 4 MHz. (Note: Changing the divider when the Fast IRC is enabled is not supported).

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting Fast Clock Internal Reference Divider Setting
-----------------	--

28.1.1.53 static uint8_t CLOCK_HAL_GetFastClkInternalRefDivider (uint32_t *baseAddr*) [inline], [static]

This function gets the Fast Clock Internal Reference Divider Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting Fast Clock Internal Reference Divider Setting

28.1.1.54 static void CLOCK_HAL_SetAutoTrimMachineCompValHigh (uint32_t *baseAddr*, uint8_t *setting*) [inline], [static]

This function sets the ATM compare value high setting. The values are used by the Auto Trim Machine to compare and adjust the Internal Reference trim values during the ATM SAR conversion.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting ATM Compare Value High Setting
-----------------	---

28.1.1.55 static uint8_t CLOCK_HAL_GetAutoTrimMachineCompValHigh (uint32_t *baseAddr*) [inline], [static]

This function gets the ATM Compare Value High Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting ATM Compare Value High Setting

28.1.1.56 static void CLOCK_HAL_SetAutoTrimMachineCompValLow (*uint32_t baseAddr*, *uint8_t setting*) [inline], [static]

This function sets the ATM compare value low setting. The values are used by the Auto Trim Machine to compare and adjust Internal Reference trim values during the ATM SAR conversion.

Parameters

<i>baseAddr</i>	Base address for current MCG instance. setting ATM Compare Value Low Setting
-----------------	--

28.1.1.57 static uint8_t CLOCK_HAL_GetAutoTrimMachineCompValLow (uint32_t *baseAddr*) [inline], [static]

This function gets the ATM Compare Value Low Setting.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

setting ATM Compare Value Low Setting

28.1.1.58 mcg_modes_t CLOCK_HAL_GetMcgMode (uint32_t *baseAddr*)

This is an internal function that checks the MCG registers and determine the current MCG mode

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

Returns

mcgMode Current MCG mode or error code mcg_modes_t

28.1.1.59 uint32_t CLOCK_HAL_GetFllFrequency (uint32_t *baseAddr*, int32_t *fllRef*)

This function calculates and checks the FLL frequency value based on input value.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
-----------------	--

MCG HAL driver

<i>fllRef</i>	- FLL reference clock in Hz.
---------------	------------------------------

Returns

value FLL output frequency (Hz) or error code

28.1.1.60 uint32_t CLOCK_HAL_SetFeiToFeeMode (uint32_t *baseAddr*, mcg_oscsel_select_t *oscselVal*, uint32_t *crystalVal*, mcg_high_gain_osc_select_t *hgoVal*, mcg_external_ref_clock_select_t *erefsvVal*)

This function transitions the MCG from FEI mode to FEE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>oscselVal</i>	- oscillator selection value 0 - OSC 0, 1 - RTC 32k, 2 - IRC 48M
<i>crystalVal</i>	- external clock frequency in Hz oscselVal - 0 erefsVal - 0: osc0 external clock frequency erefsVal - 1: osc0 crystal clock frequency oscselVal - 1: RTC 32Khz clock source frequency oscselVal - 2: IRC 48Mhz clock source frequency
<i>hgoVal</i>	- selects whether low power or high gain mode is selected for the crystal oscillator. This value is only valid when oscselVal is 0 and erefsVal is 1.
<i>erefsvVal</i>	- selects external clock (=0) or crystal OSC (=1)

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.61 uint32_t CLOCK_HAL_SetFeiToFbiMode (uint32_t *baseAddr*, uint32_t *ircFreq*, mcg_internal_ref_clock_select_t *ircSelect*)

This function transitions the MCG from FEI mode to FBI mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value
<i>ircSelect</i>	- slow or fast clock selection 0: slow, 1: fast

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.62 `uint32_t CLOCK_HAL_SetFeiToFbeMode (uint32_t baseAddr,
mcg_oscsel_select_t oscselVal, uint32_t crystalVal, mcg_high_gain_osc_select_t
hgoVal, mcg_external_ref_clock_select_t erefsVal)`

This function transitions the MCG from FEI mode to FBE mode.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>oscSelVal</i>	- oscillator selection value 0 - OSC 0, 1 - RTC 32k, 2 - IRC 48M
<i>crystalVal</i>	- external clock frequency in Hz oscSelVal - 0: erefsVal - 0: osc0 external clock frequency erefsVal - 1: osc0 crystal clock frequency oscSelVal - 1: RTC 32Khz clock source frequency oscSelVal - 2: IRC 48Mhz clock source frequency
<i>hgoVal</i>	- selects whether low power or high gain mode is selected for the crystal oscillator. This value is only valid when oscSelVal is 0 and erefsVal is 1.
<i>erefsVal</i>	- selects external clock (=0) or crystal OSC (=1)

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.63 uint32_t CLOCK_HAL_SetFeeToFeiMode (uint32_t *baseAddr*, uint32_t *ircFreq*)

This function transitions the MCG from FEE mode to FEI mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value (slow)

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.64 uint32_t CLOCK_HAL_SetFeeToFbiMode (uint32_t *baseAddr*, uint32_t *ircFreq*, mcg_internal_ref_clock_select_t *ircSelect*)

This function transitions the MCG from FEE mode to FBI mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value
<i>ircSelect</i>	- slow or fast clock selection 0: slow, 1: fast

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.65 uint32_t CLOCK_HAL_SetFeeToFbeMode (uint32_t *baseAddr*, uint32_t *crystalVal*)

This function transitions the MCG from FEE mode to FBE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external reference clock frequency value

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.66 uint32_t CLOCK_HAL_SetFbiToFeiMode (uint32_t *baseAddr*, uint32_t *ircFreq*)

This function transitions the MCG from FBI mode to FEI mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value (slow)

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.67 uint32_t CLOCK_HAL_SetFbiToFeeMode (uint32_t *baseAddr*, mcg_oscsel_select_t *oscselVal*, uint32_t *crystalVal*, mcg_high_gain_osc_select_t *hgoVal*, mcg_external_ref_clock_select_t *erefsvVal*)

This function transitions the MCG from FBI mode to FEE mode.

MCG HAL driver

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>oscselVal</i>	- oscillator selection value 0 - OSC 0, 1 - RTC 32k, 2 - IRC 48M
<i>crystalVal</i>	- external clock frequency in Hz oscselVal - 0 erefsVal - 0: osc0 external clock frequency erefsVal - 1: osc0 crystal clock frequency oscselVal - 1: RTC 32Khz clock source frequency oscselVal - 2: IRC 48Mhz clock source frequency
<i>hgoVal</i>	- selects whether low power or high gain mode is selected for the crystal oscillator. This value is only valid when oscselVal is 0 and erefsVal is 1.
<i>erefsVal</i>	- selects external clock (=0) or crystal OSC (=1)

Returns

value MCGCLKOUT frequency (Hz) or error code

**28.1.1.68 uint32_t CLOCK_HAL_SetFbiToFbeMode (uint32_t *baseAddr*,
mcg_oscsel_select_t *oscselVal*, uint32_t *crystalVal*, mcg_high_gain_osc_select_t
hgoVal, mcg_external_ref_clock_select_t *erefsVal*)**

This function transitions the MCG from FBI mode to FBE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>oscselVal</i>	- oscillator selection value 0 - OSC 0, 1 - RTC 32k, 2 - IRC 48M
<i>crystalVal</i>	- external clock frequency in Hz oscselVal - 0 erefsVal - 0: osc0 external clock frequency erefsVal - 1: osc0 crystal clock frequency oscselVal - 1: RTC 32Khz clock source frequency oscselVal - 2: IRC 48Mhz clock source frequency
<i>hgoVal</i>	- selects whether low power or high gain mode is selected for the crystal oscillator. This value is only valid when oscselVal is 0 and erefsVal is 1.
<i>erefsVal</i>	- selects external clock (=0) or crystal OSC (=1)

Returns

value MCGCLKOUT frequency (Hz) or error code

**28.1.1.69 uint32_t CLOCK_HAL_SetFbiToBlpiMode (uint32_t *baseAddr*, uint32_t *ircFreq*,
mcg_internal_ref_clock_select_t *ircSelect*)**

This function transitions the MCG from FBI mode to BLPI mode. This is achieved by setting the MCG_C2[LP] bit.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value
<i>ircSelect</i>	- slow or fast clock selection 0: slow, 1: fast

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.70 **uint32_t CLOCK_HAL_SetBlpiToFbiMode (uint32_t *baseAddr*, uint32_t *ircFreq*, uint8_t *ircSelect*)**

This function transitions the MCG from BLPI mode to FBI mode. This is achieved by clearing the MCG_C2[LP] bit.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value
<i>ircSelect</i>	- slow or fast clock selection 0: slow, 1: fast

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.71 **uint32_t CLOCK_HAL_SetFbeToFeeMode (uint32_t *baseAddr*, uint32_t *crystalVal*)**

This function transitions the MCG from FBE mode to FEE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external reference clock frequency value

Returns

value MCGCLKOUT frequency (Hz) or error code

MCG HAL driver

28.1.1.72 uint32_t CLOCK_HAL_SetFbeToFeiMode (uint32_t *baseAddr*, uint32_t *ircFreq*)

This function transitions the MCG from FBE mode to FEI mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value (slow)

Returns

value MCGCLKOUT frequency (Hz) or error code END

28.1.1.73 `uint32_t CLOCK_HAL_SetFbeToFbiMode (uint32_t baseAddr, uint32_t ircFreq, mcg_internal_ref_clock_select_t ircSelect)`

This function transitions the MCG from FBE mode to FBI mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>ircFreq</i>	- internal reference clock frequency value
<i>ircSelect</i>	- slow or fast clock selection 0: slow, 1: fast

Returns

value MCGCLKOUT frequency (Hz) or error code END

28.1.1.74 `uint32_t CLOCK_HAL_SetFbeToPbeMode (uint32_t baseAddr, uint32_t crystalVal, mcg_pll_clk_select_t pllcsSelect, uint8_t prdivVal, uint8_t vdivVal)`

This function transitions the MCG from FBE mode to PBE mode. The function requires the desired OSC and PLL be passed in to it for compatibility with the future support of OSC/PLL selection (This function presently only supports OSC0 as PLL source)

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz
<i>pllcsSelect</i>	- 0 to select PLL0, non-zero to select PLL1.

MCG HAL driver

<i>prdivVal</i>	- value to divide the external clock source by to create the desired PLL reference clock frequency
<i>vdivVal</i>	- value to multiply the PLL reference clock frequency by

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.75 uint32_t CLOCK_HAL_SetFbeToBlpeMode (*uint32_t baseAddr, uint32_t crystalVal*)

This function transitions the MCG from FBE mode to BLPE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.76 uint32_t CLOCK_HAL_SetPbeToFbeMode (*uint32_t baseAddr, uint32_t crystalVal*)

This function transitions the MCG from PBE mode to FBE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.77 uint32_t CLOCK_HAL_SetPbeToPeeMode (*uint32_t baseAddr, uint32_t crystalVal, mcg_pll_clk_select_t pllcsSelect*)

This function transitions the MCG from PBE mode to PEE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz
<i>pllcsSelect</i>	- PLLCS select setting <code>mcg_pll_clk_select_t</code> is defined in <code>fsl_mcg_hal.h</code> 0: <code>kMcgPllcsSelectPll0</code> PLL0 output clock is selected 1: <code>kMcgPllcsSelectPll1</code> PLL1 output clock is selected

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.78 `uint32_t CLOCK_HAL_SetPbeToBlpeMode (uint32_t baseAddr, uint32_t crystalVal)`

This function transitions the MCG from PBE mode to BLPE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.79 `uint32_t CLOCK_HAL_SetPeeToPbeMode (uint32_t baseAddr, uint32_t crystalVal)`

This function transitions the MCG from PEE mode to PBE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz

Returns

value MCGCLKOUT frequency (Hz) or error code

MCG HAL driver

28.1.1.80 uint32_t CLOCK_HAL_SetBlpeToPbeMode (uint32_t *baseAddr*, uint32_t *crystalVal*, mcg_pll_clk_select_t *pllcsSelect*, uint8_t *prdivVal*, uint8_t *vdivVal*)

This function transitions the MCG from BLPE mode to PBE mode. The function requires the desired OSC and PLL be passed in to it for compatibility with the future support of OSC/PLL selection (This function presently only supports OSC0 as PLL source)

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external clock frequency in Hz
<i>pllcsSelect</i>	- 0 to select PLL0, non-zero to select PLL1.
<i>prdivVal</i>	- value to divide the external clock source by to create the desired PLL reference clock frequency
<i>vdivVal</i>	- value to multiply the PLL reference clock frequency by

Returns

value MCGCLKOUT frequency (Hz) or error code

28.1.1.81 uint32_t CLOCK_HAL_SetBlpeToFbeMode (uint32_t *baseAddr*, uint32_t *crystalVal*)

This function transitions the MCG from BLPE mode to FBE mode.

Parameters

<i>baseAddr</i>	Base address for current MCG instance.
<i>crystalVal</i>	- external reference clock frequency value

Returns

value MCGCLKOUT frequency (Hz) or error code

Chapter 29 Oscillator (OSC)

The Kinetis SDK provides a HAL driver for the Oscillator (OSC) block of Kinetis devices.

Modules

- [OSC HAL driver](#)
The part describes the programming interface of the OSC HAL driver.
- [Shared OSC Types](#)
The part describes the programming interface for the OSC HAL driver.

OSC HAL driver

29.1 OSC HAL driver

The chapter describes the programming interface of the OSC HAL driver.

Enumerations

- enum `osc_capacitor_config_t` {
 `kOscCapacitor2p` = `OSC_CR_SC2P_MASK`,
 `kOscCapacitor4p` = `OSC_CR_SC4P_MASK`,
 `kOscCapacitor8p` = `OSC_CR_SC8P_MASK`,
 `kOscCapacitor16p` = `OSC_CR_SC16P_MASK` }
Oscillator capacitor load configurations.

oscillator control APIs

- void `OSC_HAL_SetExternalRefClkCmd` (`uint32_t` baseAddr, `bool` enable)
Enables the external reference clock for the oscillator.
- `bool OSC_HAL_GetExternalRefClkCmd` (`uint32_t` baseAddr)
Gets the external reference clock enable setting for the oscillator.
- void `OSC_HAL_SetExternalRefClkInStopModeCmd` (`uint32_t` baseAddr, `bool` enable)
Enables/disables the external reference clock in stop mode.
- `bool OSC_HAL_GetExternalRefClkInStopModeCmd` (`uint32_t` baseAddr)
Gets the external reference clock enable setting in stop mode.
- void `OSC_HAL_SetCapacitorCmd` (`uint32_t` baseAddr, `osc_capacitor_config_t` capacitorConfig, `bool` enable)
Enables the capacitor configuration for the oscillator.
- `bool OSC_HAL_GetCapacitorCmd` (`uint32_t` baseAddr, `osc_capacitor_config_t` capacitorConfig)
Gets the capacitor configuration for a specific oscillator.

29.1.0.82 OSC Hal Driver

Overview

The OSC module is a crystal oscillator. The module and the external crystal or resonator generate a reference clock for the MCU.

The `osc_hal` provides a set of API functions used to access the SIM registers including enable/disable external reference clock and set the capacitor configurations.

Oscillator Control Register Access API Functions

Each Oscillator has only one control register, `OSCx_CR`. It provides an external reference clock enable, external reference clock stop enable, and the capacitor load configuration settings.

Example of OSC HAL access APIs

```
#include "osc/hal/fsl_osc_hal.h"

// Enable the external reference clock
OSC_HAL_SetExternalRefClkCmd(kOsc0, true);

// Disable the external reference clock
OSC_HAL_SetExternalRefClkCmd(kOsc0, false);

#include "osc/hal/fsl_osc_hal.h"

// Enable the osc0 capacitor 2p
OSC_HAL_SetCapacitorCmd(kOsc0, kOscCapacitor2p, true);

// Disable the osc0 capacitor 16p
OSC_HAL_SetCapacitorCmd(kOsc0, kOscCapacitor16p, false);
```

29.1.1 Enumeration Type Documentation

29.1.1.1 enum osc_capacitor_config_t

Enumerator

kOscCapacitor2p 2 pF capacitor load
kOscCapacitor4p 4 pF capacitor load
kOscCapacitor8p 8 pF capacitor load
kOscCapacitor16p 16 pF capacitor load

29.1.2 Function Documentation

29.1.2.1 void OSC_HAL_SetExternalRefClkCmd (uint32_t baseAddr, bool enable)

This function enables the external reference clock output for the oscillator, OSCERCLK. This clock is used by many peripherals. It should be enabled at an early system initialization stage to ensure the peripherals can select and use it.

Parameters

<i>baseAddr</i>	Oscillator register base address
<i>enable</i>	enable/disable the clock

29.1.2.2 bool OSC_HAL_GetExternalRefClkCmd (uint32_t baseAddr)

This function gets the external reference clock output enable setting for the oscillator , OSCERCLK. This clock is used by many peripherals. It should be enabled at an early system initialization stage to ensure the peripherals could select and use it.

OSC HAL driver

Parameters

<i>baseAddr</i>	Oscillator register base address
-----------------	----------------------------------

Returns

enable clock enable/disable setting

29.1.2.3 void OSC_HAL_SetExternalRefClkInStopModeCmd (uint32_t *baseAddr*, bool *enable*)

This function enables/disables the external reference clock (OSCERCLK) when an MCU enters the stop mode.

Parameters

<i>baseAddr</i>	Oscillator register base address
<i>enable</i>	enable/disable setting

29.1.2.4 bool OSC_HAL_GetExternalRefClkInStopModeCmd (uint32_t *baseAddr*)

This function gets the external reference clock (OSCERCLK) enable setting when an MCU enters stop mode.

Parameters

<i>baseAddr</i>	Oscillator register base address
-----------------	----------------------------------

29.1.2.5 void OSC_HAL_SetCapacitorCmd (uint32_t *baseAddr*, osc_capacitor_config_t *capacitorConfig*, bool *enable*)

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

Parameters

<i>baseAddr</i>	Oscillator register base address
-----------------	----------------------------------

<i>capacitor-Config</i>	Capacitor configuration. (2p, 4p, 8p, 16p)
<i>enable</i>	enable/disable the Capacitor configuration

29.1.2.6 **bool OSC_HAL_GetCapacitorCmd (uint32_t *baseAddr*, osc_capacitor_config_t *capacitorConfig*)**

This function gets the specified capacitors configuration for an oscillator.

Parameters

<i>baseAddr</i>	Oscillator register base address
<i>capacitor-Config</i>	Capacitor configuration.

Returns

enable enable/disable setting

Shared OSC Types

29.2 Shared OSC Types

The chapter describes the programming interface for the OSC HAL driver.

Chapter 30

Power Management Controller (PMC)

The Kinetis SDK provides a HAL driver for the Power Management Controller (PMC) block of Kinetis devices.

Enumerations

- enum `pmc_low_volt_warn_volt_select_t` {
 `kPmcLowVoltWarnVoltLowTrip`,
 `kPmcLowVoltWarnVoltMid1Trip`,
 `kPmcLowVoltWarnVoltMid2Trip`,
 `kPmcLowVoltWarnVoltHighTrip` }
 Low-Voltage Warning Voltage Select.
- enum `pmc_low_volt_detect_volt_select_t` {
 `kPmcLowVoltDetectVoltLowTrip`,
 `kPmcLowVoltDetectVoltHighTrip` }
 Low-Voltage Detect Voltage Select.
- enum `pmc_int_select_t` {
 `kPmcIntLowVoltDetect`,
 `kPmcIntLowVoltWarn` }
 interrupt control

Power Management Controller Control APIs

- void `PMC_HAL_SetLowVoltIntCmd` (uint32_t baseAddr, `pmc_int_select_t` intSelect, bool enable)
 Enables/Disables low voltage-related interrupts.
- static void `PMC_HAL_SetLowVoltDetectResetCmd` (uint32_t baseAddr, bool enable)
 Low-Voltage Detect Hardware Reset Enable/Disable (write once)
- static void `PMC_HAL_SetLowVoltDetectAck` (uint32_t baseAddr)
 Low-Voltage Detect Acknowledge.
- static bool `PMC_HAL_GetLowVoltDetectFlag` (uint32_t baseAddr)
 Low-Voltage Detect Flag Read.
- static void `PMC_HAL_SetLowVoltDetectVoltMode` (uint32_t baseAddr, `pmc_low_volt_detect_volt_select_t` select)
 Sets the Low-Voltage Detect Voltage Mode.
- static `pmc_low_volt_detect_volt_select_t` `PMC_HAL_GetLowVoltDetectVoltMode` (uint32_t baseAddr)
 Gets the Low-Voltage Detect Voltage Mode.
- static void `PMC_HAL_SetLowVoltWarnAck` (uint32_t baseAddr)
 Low-Voltage Warning Acknowledge.
- static bool `PMC_HAL_GetLowVoltWarnFlag` (uint32_t baseAddr)
 Low-Voltage Warning Flag Read.
- static void `PMC_HAL_SetLowVoltWarnVoltMode` (uint32_t baseAddr, `pmc_low_volt_warn_volt_select_t` select)

Enumeration Type Documentation

- static
 - `pmc_low_volt_warn_volt_select_t` `PMC_HAL_GetLowVoltWarnVoltMode` (`uint32_t` `baseAddr`)
Gets the Low-Voltage Warning Voltage Mode.
 - static void `PMC_HAL_SetBandgapBufferCmd` (`uint32_t` `baseAddr`, `bool enable`)
Enables/Disables the Bandgap Buffer.
 - static `uint8_t` `PMC_HAL_GetAckIsolation` (`uint32_t` `baseAddr`)
Gets the acknowledge isolation value.
 - static void `PMC_HAL_SetClearAckIsolation` (`uint32_t` `baseAddr`)
Clears an acknowledge isolation.
 - static `uint8_t` `PMC_HAL_GetRegulatorStatus` (`uint32_t` `baseAddr`)
Gets the Regulator regulation status.

30.0.1 PMC HAL driver

Overview

The Power Management Controller (PMC) contains the internal voltage regulator, power on reset (POR), and low voltage detect system.

The `pmc_hal` provides a set of APIs used to access the control registers including enable/disable features provided by this module.

Power Management Controller feature access API functions

1. Internal voltage regulator
2. Active POR providing brown-out detect
3. Low-voltage detect supporting two low-voltage trip points with four warning levels per trip point

This is an example of PMC HAL access API functions.

```
#include "pmc/hal/fsl_pmc_hal.h"

// Low-Voltage Detect Interrupt Enable
PMC_HAL_SetLowVoltIntCmd(kPmcIntLowVoltDetect, true);

// Low-Voltage Detect Interrupt Disable (use polling)
PMC_HAL_SetLowVoltIntCmd(kPmcIntLowVoltDetect, false);

// Set Low-Voltage Detect Voltage Select to Low trip point (V LVD = V LVDL )
PMC_HAL_SetLowVoltDetectVoltMode(
    kPmcLowVoltDetectVoltLowTrip);
```

30.1 Enumeration Type Documentation

30.1.1 enum pmc_low_volt_warn_volt_select_t

Enumerator

- **`kPmcLowVoltWarnVoltLowTrip`** Low trip point selected (VLVW = VLVW1)
- **`kPmcLowVoltWarnVoltMid1Trip`** Mid 1 trip point selected (VLVW = VLVW2)
- **`kPmcLowVoltWarnVoltMid2Trip`** Mid 2 trip point selected (VLVW = VLVW3)

kPmcLowVoltWarnVoltHighTrip High trip point selected (VLVW = VLVW4)

30.1.2 enum pmc_low_volt_detect_volt_select_t

Enumerator

kPmcLowVoltDetectVoltLowTrip Low trip point selected (V LVD = V LVDL)

kPmcLowVoltDetectVoltHighTrip High trip point selected (V LVD = V LVDH)

30.1.3 enum pmc_int_select_t

Enumerator

kPmcIntLowVoltDetect Low Voltage Detect Interrupt.

kPmcIntLowVoltWarn Low Voltage Warning Interrupt.

30.2 Function Documentation

30.2.1 void PMC_HAL_SetLowVoltIntCmd (*uint32_t baseAddr*, *pmc_int_select_t intSelect*, *bool enable*)

This function enables the interrupt for the low voltage detection, warning, etc. When enabled, if the LVDF (Low Voltage Detect Flag) is set, a hardware interrupt occurs.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
<i>intSelect</i>	interrupt select
<i>enable</i>	enable/disable the interrupt

30.2.2 static void PMC_HAL_SetLowVoltDetectResetCmd (*uint32_t baseAddr*, *bool enable*) [inline], [static]

This function enables/disables the hardware reset for the low voltage detection. When enabled, if the LVDF (Low Voltage Detect Flag) is set, a hardware reset occurs. This setting is a write-once-only. Any additional writes are ignored.

Function Documentation

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
<i>enable</i>	enable/disable the LVD hardware reset

30.2.3 static void PMC_HAL_SetLowVoltDetectAck (*uint32_t baseAddr*) [**inline**], [**static**]

This function acknowledges the low voltage detection errors (write 1 to clear LVDF).

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

30.2.4 static bool PMC_HAL_GetLowVoltDetectFlag (*uint32_t baseAddr*) [**inline**], [**static**]

This function reads the current LVDF status. If it returns 1, a low voltage event is detected.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

Returns

status Current low voltage detect flag

- true: Low-Voltage detected
- false: Low-Voltage not detected

30.2.5 static void PMC_HAL_SetLowVoltDetectVoltMode (*uint32_t baseAddr,* *pmc_low_volt_detect_volt_select_t select*) [**inline**], [**static**]

This function sets the low voltage detect voltage select. It sets the low voltage detect trip point voltage (Vlvd). An application can select either a low-trip or a high-trip point. See a chip reference manual for details.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
<i>select</i>	Voltage select setting defined in pmc_lvdv_select_t

30.2.6 static pmc_low_volt_detect_volt_select_t PMC_HAL_GetLowVoltDetectVoltMode (uint32_t *baseAddr*) [inline], [static]

This function gets the low voltage detect voltage select. It gets the low voltage detect trip point voltage (Vlvd). An application can select either a low-trip or a high-trip point. See a chip reference manual for details.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

Returns

select Current voltage select setting

30.2.7 static void PMC_HAL_SetLowVoltWarnAck (uint32_t *baseAddr*) [inline], [static]

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

30.2.8 static bool PMC_HAL_GetLowVoltWarnFlag (uint32_t *baseAddr*) [inline], [static]

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Function Documentation

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

Returns

- status Current LVWF status
- true: Low-Voltage Warning Flag is set.
 - false: the Low-Voltage Warning does not happen.

30.2.9 static void PMC_HAL_SetLowVoltWarnVoltMode (uint32_t *baseAddr*, pmc_low_volt_warn_volt_select_t *select*) [inline], [static]

This function sets the low voltage warning voltage select. It sets the low voltage warning trip point voltage (Vlvw). An application can select either a low, mid1, mid2 and a high-trip point. See a chip reference manual for details and the pmc_lvvw_select_t for supported settings.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
<i>select</i>	Low voltage warning select setting

30.2.10 static pmc_low_volt_warn_volt_select_t PMC_HAL_GetLowVoltWarnVoltMode (uint32_t *baseAddr*) [inline], [static]

This function gets the low voltage warning voltage select. It gets the low voltage warning trip point voltage (Vlvw). See the pmc_lvvw_select_t for supported settings.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

Returns

- select* Current low voltage warning select setting

30.2.11 static void PMC_HAL_SetBandgapBufferCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]

This function enables/disables the Bandgap buffer.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
<i>enable</i>	enable/disable the Bangap Buffer.

30.2.12 static uint8_t PMC_HAL_GetAckIsolation (uint32_t *baseAddr*) [inline], [static]

This function reads the Acknowledge Isolation setting that indicates whether certain peripherals and the I/O pads are in a latched state as a result of having been in the VLLS mode.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

Returns

value ACK isolation 0 - Peripherals and I/O pads are in a normal run state. 1 - Certain peripherals and I/O pads are in an isolated and latched state.

30.2.13 static void PMC_HAL_SetClearAckIsolation (uint32_t *baseAddr*) [inline], [static]

This function clears the ACK Isolation flag. Writing one to this setting when it is set releases the I/O pads and certain peripherals to their normal run mode state.

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

30.2.14 static uint8_t PMC_HAL_GetRegulatorStatus (uint32_t *baseAddr*) [inline], [static]

This function returns the regulator to a run regulation status. It provides the current status of the internal voltage regulator.

Function Documentation

Parameters

<i>baseAddr</i>	Base address for current PMC instance.
-----------------	--

Returns

value Regulation status 0 - Regulator is in a stop regulation or in transition to/from it. 1 - Regulator is in a run regulation.

Chapter 31

Port Control and Interrupts (PORT)

The Kinetis SDK provides a HAL driver for the Port Control and Interrupts (PORT) block of Kinetis devices.

Modules

- [PORT HAL driver](#)

The part describes the programming interface of the PORT HAL driver.

POR T HAL driver

31.1 POR T HAL driver

The chapter describes the programming interface of the POR T HAL driver.

Enumerations

- enum `port_pull_t` {
 `kPortPullDown` = 0U,
 `kPortPullUp` = 1U }
 Internal resistor pull feature selection.
- enum `port_slew_rate_t` {
 `kPortFastSlewRate` = 0U,
 `kPortSlowSlewRate` = 1U }
 Slew rate selection.
- enum `port_drive_strength_t` {
 `kPortLowDriveStrength` = 0U,
 `kPortHighDriveStrength` = 1U }
 Configures the drive strength.
- enum `port_mux_t` {
 `kPortPinDisabled` = 0U,
 `kPortMuxAsGpio` = 1U,
 `kPortMuxAlt2` = 2U,
 `kPortMuxAlt3` = 3U,
 `kPortMuxAlt4` = 4U,
 `kPortMuxAlt5` = 5U,
 `kPortMuxAlt6` = 6U,
 `kPortMuxAlt7` = 7U }
 Pin mux selection.
- enum `port_interrupt_config_t` {
 `kPortIntDisabled` = 0x0U,
 `kPortDmaRisingEdge` = 0x1U,
 `kPortDmaFallingEdge` = 0x2U,
 `kPortDmaEitherEdge` = 0x3U,
 `kPortIntLogicZero` = 0x8U,
 `kPortIntRisingEdge` = 0x9U,
 `kPortIntFallingEdge` = 0xAU,
 `kPortIntEitherEdge` = 0xBU,
 `kPortIntLogicOne` = 0xCU }
 Digital filter clock source selection.

Configuration

- static void `POR T _HAL_SetPullMode` (uint32_t baseAddr, uint32_t pin, `port_pull_t` pullSelect)
 Selects the internal resistor as pull-down or pull-up.
- static void `POR T _HAL_SetPullCmd` (uint32_t baseAddr, uint32_t pin, bool isPullEnabled)

- Enables or disables the internal pull resistor.
- static void **PORT_HAL_SetSlewRateMode** (uint32_t baseAddr, uint32_t pin, **port_slew_rate_t** rateSelect)

Configures the fast/slow slew rate if the pin is used as a digital output.
- static void **PORT_HAL_SetPassiveFilterCmd** (uint32_t baseAddr, uint32_t pin, bool isPassiveFilterEnabled)

Configures the passive filter if the pin is used as a digital input.
- static void **PORT_HAL_SetDriveStrengthMode** (uint32_t baseAddr, uint32_t pin, **port_drive_strength_t** driveSelect)

Configures the drive strength if the pin is used as a digital output.
- static void **PORT_HAL_SetMuxMode** (uint32_t baseAddr, uint32_t pin, **port_mux_t** mux)

Configures the pin muxing.
- void **PORT_HAL_SetLowGlobalPinCtrl** (uint32_t baseAddr, uint16_t lowPinSelect, uint16_t config)

Configures the low half of the pin control register for the same settings.
- void **PORT_HAL_SetHighGlobalPinCtrl** (uint32_t baseAddr, uint16_t highPinSelect, uint16_t config)

Configures the high half of pin control register for the same settings.

Interrupt

- static void **PORT_HAL_SetPinIntMode** (uint32_t baseAddr, uint32_t pin, **port_interrupt_config_t** intConfig)

Configures the port pin interrupt/DMA request.
- static **port_interrupt_config_t** **PORT_HAL_GetPinIntMode** (uint32_t baseAddr, uint32_t pin)

Gets the current port pin interrupt/DMA request configuration.
- static bool **PORT_HAL_IsPinIntPending** (uint32_t baseAddr, uint32_t pin)

Reads the individual pin-interrupt status flag.
- static void **PORT_HAL_ClearPinIntFlag** (uint32_t baseAddr, uint32_t pin)

Clears the individual pin-interrupt status flag.
- static uint32_t **PORT_HAL_GetPortIntFlag** (uint32_t baseAddr)

Reads the entire port interrupt status flag.
- static void **PORT_HAL_ClearPortIntFlag** (uint32_t baseAddr)

Clears the entire port interrupt status flag.

31.1.0.1 PORT HAL Driver

Overview

Port control and interrupts hardware driver configuration. Use these functions to set port control and external interrupt functions. Most functions can be configured independently for each pin in the 32-bit port and affect the pin regardless of its pin muxing state. To use these functions, pass into instance number (HW_PORTA, HW_PORTB, HW_PORTC, etc).

31.1.1 Enumeration Type Documentation

31.1.1.1 enum port_pull_t

Enumerator

kPortPullDown internal pull-down resistor is enabled.

kPortPullUp internal pull-up resistor is enabled.

31.1.1.2 enum port_slew_rate_t

Enumerator

kPortFastSlewRate fast slew rate is configured.

kPortSlowSlewRate slow slew rate is configured.

31.1.1.3 enum port_drive_strength_t

Enumerator

kPortLowDriveStrength low drive strength is configured.

kPortHighDriveStrength high drive strength is configured.

31.1.1.4 enum port_mux_t

Enumerator

kPortPinDisabled corresponding pin is disabled as analog.

kPortMuxAsGpio corresponding pin is configured as GPIO.

kPortMuxAlt2 chip-specific

kPortMuxAlt3 chip-specific

kPortMuxAlt4 chip-specific

kPortMuxAlt5 chip-specific

kPortMuxAlt6 chip-specific

kPortMuxAlt7 chip-specific

31.1.1.5 enum port_interrupt_config_t

Configures the interrupt generation condition.

Enumerator

kPortIntDisabled Interrupt/DMA request is disabled.

kPortDmaRisingEdge DMA request on rising edge.
kPortDmaFallingEdge DMA request on falling edge.
kPortDmaEitherEdge DMA request on either edge.
kPortIntLogicZero Interrupt when logic zero.
kPortIntRisingEdge Interrupt on rising edge.
kPortIntFallingEdge Interrupt on falling edge.
kPortIntEitherEdge Interrupt on either edge.
kPortIntLogicOne Interrupt when logic one.

31.1.2 Function Documentation

31.1.2.1 static void PORT_HAL_SetPullMode (*uint32_t baseAddr*, *uint32_t pin*, *port_pull_t pullSelect*) [inline], [static]

Pull configuration is valid in all digital pin muxing modes.

Parameters

<i>baseAddr</i>	port base address.
<i>pin</i>	port pin number
<i>pullSelect</i>	internal resistor pull feature selection <ul style="list-style-type: none"> • <i>kPortPullDown</i>: internal pull-down resistor is enabled. • <i>kPortPullUp</i> : internal pull-up resistor is enabled.

31.1.2.2 static void PORT_HAL_SetPullCmd (*uint32_t baseAddr*, *uint32_t pin*, *bool isPullEnabled*) [inline], [static]

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number
<i>isPullEnabled</i>	internal pull resistor enable or disable <ul style="list-style-type: none"> • true : internal pull resistor is enabled. • false: internal pull resistor is disabled.

31.1.2.3 static void PORT_HAL_SetSlewRateMode (*uint32_t baseAddr*, *uint32_t pin*, *port_slew_rate_t rateSelect*) [inline], [static]

POR T HAL driver

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number
<i>rateSelect</i>	slew rate selection <ul style="list-style-type: none">• kPortFastSlewRate: fast slew rate is configured.• kPortSlowSlewRate: slow slew rate is configured.

31.1.2.4 static void PORT_HAL_SetPassiveFilterCmd (uint32_t *baseAddr*, uint32_t *pin*, bool *isPassiveFilterEnabled*) [inline], [static]

If enabled, a low pass filter (10 MHz to 30 MHz bandwidth) is enabled on the digital input path. Disable the Passive Input Filter when supporting high speed interfaces (> 2 MHz) on the pin.

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number
<i>isPassiveFilterEnabled</i>	passive filter configuration <ul style="list-style-type: none">• false: passive filter is disabled.• true : passive filter is enabled.

31.1.2.5 static void PORT_HAL_SetDriveStrengthMode (uint32_t *baseAddr*, uint32_t *pin*, port_drive_strength_t *driveSelect*) [inline], [static]

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number
<i>driveSelect</i>	drive strength selection <ul style="list-style-type: none">• kLowDriveStrength : low drive strength is configured.• kHighDriveStrength: high drive strength is configured.

31.1.2.6 static void PORT_HAL_SetMuxMode (uint32_t *baseAddr*, uint32_t *pin*, port_mux_t *mux*) [inline], [static]

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number
<i>mux</i>	pin muxing slot selection <ul style="list-style-type: none"> • kPinDisabled: Pin disabled. • kMuxAsGpio : Set as GPIO. • others : chip-specific.

31.1.2.7 void PORT_HAL_SetLowGlobalPinCtrl (*uint32_t baseAddr, uint16_t lowPinSelect, uint16_t config*)

This function operates pin 0 -15 of one specific port.

Parameters

<i>baseAddr</i>	port base address
<i>lowPinSelect</i>	update corresponding pin control register or not. For a specific bit: <ul style="list-style-type: none"> • 0: corresponding low half of pin control register won't be updated according to configuration. • 1: corresponding low half of pin control register will be updated according to configuration.
<i>config</i>	value is written to a low half port control register bits[15:0].

31.1.2.8 void PORT_HAL_SetHighGlobalPinCtrl (*uint32_t baseAddr, uint16_t highPinSelect, uint16_t config*)

This function operates pin 16 -31 of one specific port.

Parameters

<i>baseAddr</i>	port base address
<i>highPinSelect</i>	update corresponding pin control register or not. For a specific bit: <ul style="list-style-type: none"> • 0: corresponding high half of pin control register won't be updated according to configuration. • 1: corresponding high half of pin control register will be updated according to configuration.
<i>config</i>	value is written to a high half port control register bits[15:0].

31.1.2.9 static void PORT_HAL_SetPinIntMode (uint32_t baseAddr, uint32_t pin, port_interrupt_config_t intConfig) [inline], [static]

Parameters

<i>baseAddr</i>	port base address.
<i>pin</i>	port pin number
<i>intConfig</i>	interrupt configuration <ul style="list-style-type: none">• kIntDisabled : Interrupt/DMA request disabled.• kDmaRisingEdge : DMA request on rising edge.• kDmaFallingEdge: DMA request on falling edge.• kDmaEitherEdge : DMA request on either edge.• KIntLogicZero : Interrupt when logic zero.• KIntRisingEdge : Interrupt on rising edge.• KIntFallingEdge: Interrupt on falling edge.• KIntEitherEdge : Interrupt on either edge.• KIntLogicOne : Interrupt when logic one.

31.1.2.10 static port_interrupt_config_t PORT_HAL_GetPinIntMode (uint32_t baseAddr, uint32_t pin) [inline], [static]

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number

Returns

interrupt configuration

- kIntDisabled : Interrupt/DMA request disabled.
- kDmaRisingEdge : DMA request on rising edge.
- kDmaFallingEdge: DMA request on falling edge.
- kDmaEitherEdge : DMA request on either edge.
- KIntLogicZero : Interrupt when logic zero.
- KIntRisingEdge : Interrupt on rising edge.
- KIntFallingEdge: Interrupt on falling edge.
- KIntEitherEdge : Interrupt on either edge.
- KIntLogicOne : Interrupt when logic one.

31.1.2.11 static bool PORT_HAL_IsPinIntPending (*uint32_t baseAddr, uint32_t pin*) [inline], [static]

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

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number

Returns

current pin interrupt status flag
• 0: interrupt is not detected.
• 1: interrupt is detected.

31.1.2.12 static void PORT_HAL_ClearPinIntFlag (*uint32_t baseAddr, uint32_t pin*) [inline], [static]

Parameters

<i>baseAddr</i>	port base address
<i>pin</i>	port pin number

31.1.2.13 static uint32_t PORT_HAL_GetPortIntFlag (*uint32_t baseAddr*) [inline], [static]

Parameters

<i>baseAddr</i>	port base address
-----------------	-------------------

Returns

all 32 pin interrupt status flags. For specific bit:
• 0: interrupt is not detected.
• 1: interrupt is detected.

31.1.2.14 static void PORT_HAL_ClearPortIntFlag (*uint32_t baseAddr*) [inline], [static]

POR T HAL driver

Parameters

<i>baseAddr</i>	port base address
-----------------	-------------------

Chapter 32 reset control module (RCM)

The Kinetis SDK provides a HAL driver for the reset control module (RCM) block of Kinetis devices.

Enumerations

- enum `rcm_source_names_t`
System Reset Source Name definitions.
- enum `rcm_filter_run_wait_modes_t`
Reset pin filter select in Run and Wait modes.

Functions

- bool `RCM_HAL_GetSrcStatusCmd` (uint32_t baseAddr, `rcm_source_names_t` srcName)
Gets the reset source status.
- static void `RCM_HAL_SetFilterStopModeCmd` (uint32_t baseAddr, bool enable)
Sets the reset pin filter in stop mode.
- static bool `RCM_HAL_GetFilterStopModeCmd` (uint32_t baseAddr)
Gets the reset pin filter in stop mode.
- static void `RCM_HAL_SetFilterRunWaitMode` (uint32_t baseAddr, `rcm_filter_run_wait_modes_t` mode)
Sets the reset pin filter in run and wait mode.
- static `rcm_filter_run_wait_modes_t` `RCM_HAL_GetFilterRunWaitMode` (uint32_t baseAddr)
Gets the reset pin filter for stop mode.
- static void `RCM_HAL_SetFilterWidth` (uint32_t baseAddr, uint32_t width)
Sets the reset pin filter width.
- static uint32_t `RCM_HAL_GetFilterWidth` (uint32_t baseAddr)
Gets the reset pin filter for stop mode.
- static bool `RCM_HAL_GetEasyPortModeStatusCmd` (uint32_t baseAddr)
Gets the EZP_MS_B pin assert status.

32.0.3 RCM HAL driver

Overview

The RCM implements many of the reset functions for the MCU.

See the chip reset chapter for more information.

This is an example of the RCM HAL access API functions.

```
#include "rcm/hal/fsl_rcm_hal.h"

printf("\r\nSet the filter control reg for stop mode to LPO\r\n\r\n");
RCM_HAL_SetFilterStopModeCmd(true);
```

Function Documentation

```
printf("\r\nSet the filter control reg for run/wait mode to LPO\r\n\r\n");
RCM_HAL_SetFilterRunWaitMode(kRcmFilterLpoClk);
```

32.1 Function Documentation

32.1.1 **bool RCM_HAL_GetSrcStatusCmd (uint32_t *baseAddr*, rcm_source_names_t *srcName*)**

This function gets the current reset source status for a specified source.

Parameters

<i>baseAddr</i>	Register base address of RCM
<i>srcName</i>	reset source name

Returns

status true or false for specified reset source

32.1.2 **static void RCM_HAL_SetFilterStopModeCmd (uint32_t *baseAddr*, bool *enable*) [inline], [static]**

This function sets the reset pin filter enable setting in stop mode.

Parameters

<i>baseAddr</i>	Register base address of RCM
<i>enable</i>	enable or disable the filter in stop mode

32.1.3 **static bool RCM_HAL_GetFilterStopModeCmd (uint32_t *baseAddr*) [inline], [static]**

This function gets the reset pin filter enable setting in stop mode.

Parameters

<i>baseAddr</i>	Register base address of RCM
-----------------	------------------------------

Returns

enable true/false to enable or disable the filter in stop mode

**32.1.4 static void RCM_HAL_SetFilterRunWaitMode (uint32_t *baseAddr*,
rcm_filter_run_modes_t *mode*) [inline], [static]**

This function sets the reset pin filter enable setting in run/wait mode.

Function Documentation

Parameters

<i>baseAddr</i>	Register base address of RCM
<i>mode</i>	to be set for reset filter in run/wait mode

32.1.5 static rcm_filter_run_wait_modes_t RCM_HAL_GetFilterRunWaitMode (uint32_t *baseAddr*) [inline], [static]

This function gets the reset pin filter enable setting for stop mode.

Parameters

<i>baseAddr</i>	Register base address of RCM
-----------------	------------------------------

Returns

mode for reset filter in run/wait mode

32.1.6 static void RCM_HAL_SetFilterWidth (uint32_t *baseAddr*, uint32_t *width*) [inline], [static]

This function sets the reset pin filter width.

Parameters

<i>baseAddr</i>	Register base address of RCM
<i>width</i>	to be set for reset filter width

32.1.7 static uint32_t RCM_HAL_GetFilterWidth (uint32_t *baseAddr*) [inline], [static]

This function gets the reset pin filter width.

Parameters

<i>baseAddr</i>	Register base address of RCM
-----------------	------------------------------

Returns

width reset filter width

32.1.8 static bool RCM_HAL_GetEasyPortModeStatusCmd (uint32_t *baseAddr*) [inline], [static]

This function gets the easy port mode status (EZP_MS_B) pin assert status.

Parameters

<i>baseAddr</i>	Register base address of RCM
-----------------	------------------------------

Returns

status true - asserted, false - reasserted

Function Documentation

Chapter 33

System Integration Module (SIM)

The Kinetis SDK provides a HAL driver for the System Integration Module (SIM) block of Kinetis devices.

Modules

- [SIM HAL driver](#)

The part describes the programming interface of the SIM HAL driver.

SIM HAL driver

33.1 SIM HAL driver

The chapter describes the programming interface of the SIM HAL driver.

Data Structures

- struct [clock_name_config_t](#)
Clock name configuration table structure. [More...](#)

Enumerations

- enum [clock_source_names_t](#)
Clock source and sel names.
- enum [clock_divider_names_t](#)
Clock Divider names.
- enum [sim_usbsstby_stop_t](#)
SIM USB voltage regulator in standby mode setting during stop modes.
- enum [sim_usbvstby_stop_t](#)
SIM USB voltage regulator in standby mode setting during VLPR and VLPW modes.
- enum [sim_cmtuartpad_strengt_h_t](#)
SIM CMT/UART pad drive strength.
- enum [sim_ptd7pad_strengt_h_t](#)
SIM PTD7 pad drive strength.
- enum [sim_flexbus_security_level_t](#)
SIM FlexBus security level.
- enum [sim_pretrgsel_t](#)
SIM ADCx pre-trigger select.
- enum [sim_trgsel_t](#)
SIM ADCx trigger select.
- enum [sim_uart_rxsrc_t](#)
SIM receive data source select.
- enum [sim_uart_txsrc_t](#)
SIM transmit data source select.
- enum [sim_ftm_trg_src_t](#)
SIM FlexTimer x trigger y select.
- enum [sim_ftm_clk_sel_t](#)
SIM FlexTimer external clock select.
- enum [sim_ftm_ch_src_t](#)
SIM FlexTimer x channel y input capture source select.
- enum [sim_ftm_flt_sel_t](#)
SIM FlexTimer x Fault y select.
- enum [sim_tpm_clk_sel_t](#)
SIM Timer/PWM external clock select.
- enum [sim_tpm_ch_src_t](#)
SIM Timer/PWM x channel y input capture source select.
- enum [sim_hal_status_t](#)
SIM HAL API return status.
- enum [sim_usb_clock_source_t](#)
SIM USB clock source.

- enum [sim_lpuart_clock_source_t](#)
SIM LPUART clock source.
- enum [sim_pllfl_clock_sel_t](#)
SIM PLLFLLSEL clock source select.
- enum [sim_osc32k_clock_sel_t](#)
SIM OSC32KSEL clock source select.
- enum [sim_trace_clock_sel_t](#)
SIM TRACESEL clock source select.
- enum [sim_clkout_clock_sel_t](#)
SIM CLKOUT_SEL clock source select.
- enum [sim_rtclkout_clock_sel_t](#)
SIM RTCCLKOUTSEL clock source select.
- enum [sim_usb_clock_source_t](#)
SIM USB clock source.
- enum [sim_lpuart_clock_source_t](#)
SIM LPUART clock source.
- enum [sim_pllfl_clock_sel_t](#)
SIM PLLFLLSEL clock source select.
- enum [sim_osc32k_clock_sel_t](#)
SIM OSC32KSEL clock source select.
- enum [sim_trace_clock_sel_t](#)
SIM TRACESEL clock source select.
- enum [sim_clkout_clock_sel_t](#)
SIM CLKOUT_SEL clock source select.
- enum [sim_rtclkout_clock_sel_t](#)
SIM RTCCLKOUTSEL clock source select.
- enum [sim_usb_clock_source_t](#)
SIM USB clock source.
- enum [sim_lpuart_clock_source_t](#)
SIM LPUART clock source.
- enum [sim_pllfl_clock_sel_t](#)
SIM PLLFLLSEL clock source select.
- enum [sim_osc32k_clock_sel_t](#)
SIM OSC32KSEL clock source select.
- enum [sim_trace_clock_sel_t](#)
SIM TRACESEL clock source select.
- enum [sim_clkout_clock_sel_t](#)
SIM CLKOUT_SEL clock source select.
- enum [sim_rtclkout_clock_sel_t](#)
SIM RTCCLKOUTSEL clock source select.
- enum [sim_sdhc_clock_source_t](#)
SIM SDHC clock source.
- enum [sim_usb_clock_source_t](#)
SIM USB clock source.
- enum [sim_pllfl_clock_sel_t](#)
SIM PLLFLLSEL clock source select.
- enum [sim_osc32k_clock_sel_t](#)
SIM OSC32KSEL clock source select.
- enum [sim_trace_clock_sel_t](#)
SIM TRACESEL clock source select.
- enum [sim_clkout_clock_sel_t](#)
SIM CLKOUT_SEL clock source select.

SIM HAL driver

- enum [sim_rtclkout_clock_sel_t](#)
 - *SIM CLKOUT_SEL clock source select.*
- enum [sim_sdhc_clock_source_t](#)
 - *SIM SDHC clock source.*
- enum [sim_time_clock_source_t](#)
 - *SIM TIME clock source.*
- enum [sim_rmii_clock_source_t](#)
 - *SIM RMII clock source.*
- enum [sim_usb_clock_source_t](#)
 - *SIM USB clock source.*
- enum [sim_pllfl_clock_sel_t](#)
 - *SIM PLLFLLSEL clock source select.*
- enum [sim_osc32k_clock_sel_t](#)
 - *SIM OSC32KSEL clock source select.*
- enum [sim_trace_clock_sel_t](#)
 - *SIM TRACESEL clock source select.*
- enum [sim_clkout_clock_sel_t](#)
 - *SIM CLKOUT_SEL clock source select.*
- enum [sim_rtclkout_clock_sel_t](#)
 - *SIM RTCCLKOUTSEL clock source select.*
- enum [sim_sdhc_clock_source_t](#)
 - *SIM SDHC clock source.*
- enum [sim_time_clock_source_t](#)
 - *SIM TIME clock source.*
- enum [sim_rmii_clock_source_t](#)
 - *SIM RMII clock source.*
- enum [sim_usb_clock_source_t](#)
 - *SIM USB clock source.*
- enum [sim_pllfl_clock_sel_t](#)
 - *SIM PLLFLLSEL clock source select.*
- enum [sim_osc32k_clock_sel_t](#)
 - *SIM OSC32KSEL clock source select.*
- enum [sim_trace_clock_sel_t](#)
 - *SIM TRACESEL clock source select.*
- enum [sim_clkout_clock_sel_t](#)
 - *SIM CLKOUT_SEL clock source select.*
- enum [sim_rtclkout_clock_sel_t](#)
 - *SIM RTCCLKOUTSEL clock source select.*
- enum [sim_tpm_clock_source_t](#)
 - *SIM TPM clock source.*
- enum [sim_uart0_clock_source_t](#)
 - *SIM UART0 clock source.*
- enum [sim_usb_clock_source_t](#)
 - *SIM USB clock source.*
- enum [sim_pllfl_clock_sel_t](#)
 - *SIM PLLFLLSEL clock source select.*
- enum [sim_osc32k_clock_sel_t](#)
 - *SIM OSC32KSEL clock source select.*
- enum [sim_trace_clock_sel_t](#)
 - *SIM TRACESEL clock source select.*
- enum [sim_clkout_clock_sel_t](#)
 - *SIM CLKOUT_SEL clock source select.*

- enum `sim_rtclkout_clock_sel_t`
SIM RTCCLKOUTSEL clock source select.
- enum `sim_lpuart_clock_source_t`
SIM LPUART clock source.
- enum `sim_pllfl_clock_sel_t`
SIM PLLFLLSEL clock source select.
- enum `sim_osc32k_clock_sel_t`
SIM OSC32KSEL clock source select.
- enum `sim_trace_clock_sel_t`
SIM TRACESEL clock source select.
- enum `sim_clkout_clock_sel_t`
SIM CLKOUT_SEL clock source select.
- enum `sim_lpuart_clock_source_t`
SIM LPUART clock source.
- enum `sim_pllfl_clock_sel_t`
SIM PLLFLLSEL clock source select.
- enum `sim_osc32k_clock_sel_t`
SIM OSC32KSEL clock source select.
- enum `sim_trace_clock_sel_t`
SIM TRACESEL clock source select.
- enum `sim_clkout_clock_sel_t`
SIM CLKOUT_SEL clock source select.
- enum `sim_lpuart_clock_source_t`
SIM LPUART clock source.
- enum `sim_pllfl_clock_sel_t`
SIM PLLFLLSEL clock source select.
- enum `sim_osc32k_clock_sel_t`
SIM OSC32KSEL clock source select.
- enum `sim_trace_clock_sel_t`
SIM TRACESEL clock source select.
- enum `sim_clkout_clock_sel_t`
SIM CLKOUT_SEL clock source select.

Variables

- const `clock_name_config_t kClockNameConfigTable []`
clock name configuration table for specified CPU defined in `fsl_clock_module_names_Kxxx.h`

clock-related feature APIs

- `sim_hal_status_t CLOCK_HAL_SetSource` (`uint32_t baseAddr, clock_source_names_t clockSource, uint8_t setting)`
Sets the clock source setting.
- `sim_hal_status_t CLOCK_HAL_GetSource` (`uint32_t baseAddr, clock_source_names_t clockSource, uint8_t *setting)`
Gets the clock source setting.
- `sim_hal_status_t CLOCK_HAL_SetDivider` (`uint32_t baseAddr, clock_divider_names_t clockDivider, uint32_t setting)`

SIM HAL driver

- void **CLOCK_HAL_SetOutDividers** (uint32_t baseAddr, uint32_t outdiv1, uint32_t outdiv2, uint32_t outdiv3, uint32_t outdiv4)
Sets the clock out dividers setting.
- sim_hal_status_t **CLOCK_HAL_GetDivider** (uint32_t baseAddr, **clock_divider_names_t** clockDivider, uint32_t *setting)
Gets the clock divider setting.

Individual field access APIs

- void **SIM_HAL_SetAdcAlternativeTriggerCmd** (uint32_t baseAddr, uint8_t instance, bool enable)
Sets the ADCx alternate trigger enable setting.
- bool **SIM_HAL_GetAdcAlternativeTriggerCmd** (uint32_t baseAddr, uint8_t instance)
Gets the ADCx alternate trigger enable setting.
- void **SIM_HAL_SetAdcPreTriggerMode** (uint32_t baseAddr, uint8_t instance, **sim_pretrgsel_t** select)
Sets the ADCx pre-trigger select setting.
- **sim_pretrgsel_t SIM_HAL_GetAdcPreTriggerMode** (uint32_t baseAddr, uint8_t instance)
Gets the ADCx pre-trigger select setting.
- void **SIM_HAL_SetAdcTriggerMode** (uint32_t baseAddr, uint8_t instance, **sim_trgsel_t** select)
Sets the ADCx trigger select setting.
- **sim_trgsel_t SIM_HAL_GetAdcTriggerMode** (uint32_t baseAddr, uint8_t instance)
Gets the ADCx trigger select setting.
- void **SIM_HAL_SetUartRxSrcMode** (uint32_t baseAddr, uint8_t instance, **sim_uart_rxsrc_t** select)
Sets the UARTx receive data source select setting.
- **sim_uart_rxsrc_t SIM_HAL_GetUartRxSrcMode** (uint32_t baseAddr, uint8_t instance)
Gets the UARTx receive data source select setting.
- void **SIM_HAL_SetUartTxSrcMode** (uint32_t baseAddr, uint8_t instance, **sim_uart_txsrc_t** select)
Sets the UARTx transmit data source select setting.
- **sim_uart_txsrc_t SIM_HAL_GetUartTxSrcMode** (uint32_t baseAddr, uint8_t instance)
Gets the UARTx transmit data source select setting.
- static uint32_t **SIM_HAL_GetPinCntId** (uint32_t baseAddr)
Gets the Kinetis Pincount ID in System Device ID register (SIM_SDID).
- static uint32_t **SIM_HAL_GetRevId** (uint32_t baseAddr)
Gets the Kinetis Revision ID in the System Device ID register (SIM_SDID).
- static uint32_t **SIM_HAL_GetProgramFlashSize** (uint32_t baseAddr)
Gets the program flash size in the Flash Configuration Register 1 (SIM_FCFG).

IP related clock feature APIs

- void **SIM_HAL_EnableDmaClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for DMA module.
- void **SIM_HAL_DisableDmaClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for DMA module.
- bool **SIM_HAL_GetDmaGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for DMA module.
- void **SIM_HAL_EnableDmamuxClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for DMAMUX module.

- void **SIM_HAL_DisableDmamuxClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for DMAMUX module.
- bool **SIM_HAL_GetDmamuxGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for DMAMUX module.
- void **SIM_HAL_EnablePortClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for PORT module.
- void **SIM_HAL_DisablePortClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for PORT module.
- bool **SIM_HAL_GetPortGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for PORT module.
- void **SIM_HAL_EnableEwmClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for EWM module.
- void **SIM_HAL_DisableEwmClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for EWM module.
- bool **SIM_HAL_GetEwmGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for EWM module.
- void **SIM_HAL_EnableFtfClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for FTF module.
- void **SIM_HAL_DisableFtfClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for FTF module.
- bool **SIM_HAL_GetFtfGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for FTF module.
- void **SIM_HAL_EnableCrcClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for CRC module.
- void **SIM_HAL_DisableCrcClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for CRC module.
- bool **SIM_HAL_GetCrcGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for CRC module.
- void **SIM_HAL_EnableAdcClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for ADC module.
- void **SIM_HAL_DisableAdcClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for ADC module.
- bool **SIM_HAL_GetAdcGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for ADC module.
- void **SIM_HAL_EnableCmpClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for CMP module.
- void **SIM_HAL_DisableCmpClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for CMP module.
- bool **SIM_HAL_GetCmpGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for CMP module.
- void **SIM_HAL_EnableDacClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for DAC module.
- void **SIM_HAL_DisableDacClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for DAC module.
- bool **SIM_HAL_GetDacGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for DAC module.
- void **SIM_HAL_EnableVrefClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for VREF module.
- void **SIM_HAL_DisableVrefClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for VREF module.
- bool **SIM_HAL_GetVrefGateCmd** (uint32_t baseAddr, uint32_t instance)

SIM HAL driver

Get the the clock gate state for VREF module.

- void **SIM_HAL_EnableSaiClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for SAI module.
- void **SIM_HAL_DisableSaiClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for SAI module.
- bool **SIM_HAL_GetSaiGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for SAI module.
- void **SIM_HAL_EnablePdbClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for PDB module.
- void **SIM_HAL_DisablePdbClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for PDB module.
- bool **SIM_HAL_GetPdbGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for PDB module.
- void **SIM_HAL_EnableFtmClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for FTM module.
- void **SIM_HAL_DisableFtmClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for FTM module.
- bool **SIM_HAL_GetFtmGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for FTM module.
- void **SIM_HAL_EnablePitClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for PIT module.
- void **SIM_HAL_DisablePitClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for PIT module.
- bool **SIM_HAL_GetPitGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for PIT module.
- void **SIM_HAL_EnableLptimerClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for LPTIMER module.
- void **SIM_HAL_DisableLptimerClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for LPTIMER module.
- bool **SIM_HAL_GetLptimerGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for LPTIMER module.
- void **SIM_HAL_EnableRtcClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for RTC module.
- void **SIM_HAL_DisableRtcClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for RTC module.
- bool **SIM_HAL_GetRtcGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for RTC module.
- void **SIM_HAL_EnableUsbClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for USBFS module.
- void **SIM_HAL_DisableUsbClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for USBFS module.
- bool **SIM_HAL_GetUsbGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for USB module.
- void **SIM_HAL_EnableSpiClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for SPI module.
- void **SIM_HAL_DisableSpiClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for SPI module.
- bool **SIM_HAL_GetSpiGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for SPI module.
- void **SIM_HAL_EnableI2cClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for I2C module.

- void **SIM_HAL_DisableI2cClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for I2C module.
- bool **SIM_HAL_GetI2cGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for I2C module.
- void **SIM_HAL_EnableUartClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for UART module.
- void **SIM_HAL_DisableUartClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for UART module.
- bool **SIM_HAL_GetUartGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for UART module.
- void **SIM_HAL_EnableLpuartClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for LPUART module.
- void **SIM_HAL_DisableLpuartClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for LPUART module.
- bool **SIM_HAL_GetLpuartGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for LPUART module.

IP related clock feature APIs

- void **SIM_HAL_EnableRngaClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for RNGA module.
- void **SIM_HAL_DisableRngaClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for RNGA module.
- bool **SIM_HAL_GetRngaGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for RNGA module.

IP related clock feature APIs

- void **SIM_HAL_EnableFlexbusClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for FLEXBUS module.
- void **SIM_HAL_DisableFlexbusClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for FLEXBUS module.
- bool **SIM_HAL_GetFlexbusGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for FLEXBUS module.

IP related clock feature APIs

- void **SIM_HAL_EnableMpuClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for MPU module.
- void **SIM_HAL_DisableMpuClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for MPU module.
- bool **SIM_HAL_GetMpuGateCmd** (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for MPU module.
- void **SIM_HAL_EnableCmtClock** (uint32_t baseAddr, uint32_t instance)
Enable the clock for CMT module.
- void **SIM_HAL_DisableCmtClock** (uint32_t baseAddr, uint32_t instance)
Disable the clock for CMT module.

SIM HAL driver

- bool [SIM_HAL_GetCmtGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for CMT module.
- void [SIM_HAL_EnableUsbdcdClock](#) (uint32_t baseAddr, uint32_t instance)
Enable the clock for USBDCD module.
- void [SIM_HAL_DisableUsbdcdClock](#) (uint32_t baseAddr, uint32_t instance)
Disable the clock for USBDCD module.
- bool [SIM_HAL_GetUsbdcdGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for USBDCD module.
- void [SIM_HAL_EnableFlexcanClock](#) (uint32_t baseAddr, uint32_t instance)
Enable the clock for FLEXCAN module.
- void [SIM_HAL_DisableFlexcanClock](#) (uint32_t baseAddr, uint32_t instance)
Disable the clock for FLEXCAN module.
- bool [SIM_HAL_GetFlexcanGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for FLEXCAN module.
- void [SIM_HAL_EnableSdhcClock](#) (uint32_t baseAddr, uint32_t instance)
Enable the clock for SDHC module.
- void [SIM_HAL_DisableSdhcClock](#) (uint32_t baseAddr, uint32_t instance)
Disable the clock for SDHC module.
- bool [SIM_HAL_GetSdhcGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for SDHC module.

IP related clock feature APIs

- void [SIM_HAL_EnableEnetClock](#) (uint32_t baseAddr, uint32_t instance)
Enable the clock for ENET module.
- void [SIM_HAL_DisableEnetClock](#) (uint32_t baseAddr, uint32_t instance)
Disable the clock for ENET module.
- bool [SIM_HAL_GetEnetGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for ENET module.

IP related clock feature APIs

- void [SIM_HAL_EnableTpmClock](#) (uint32_t baseAddr, uint32_t instance)
Enable the clock for TPM module.
- void [SIM_HAL_DisableTpmClock](#) (uint32_t baseAddr, uint32_t instance)
Disable the clock for TPM module.
- bool [SIM_HAL_GetTpmGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for TPM module.
- void [SIM_HAL_EnableTsiClock](#) (uint32_t baseAddr, uint32_t instance)
Enable the clock for TSI module.
- void [SIM_HAL_DisableTsiClock](#) (uint32_t baseAddr, uint32_t instance)
Disable the clock for TSI module.
- bool [SIM_HAL_GetTsiGateCmd](#) (uint32_t baseAddr, uint32_t instance)
Get the the clock gate state for TSI module.

33.1.0.1 SIM Hal Driver

Overview

The system integration module (SIM) provides the system control and chip configuration registers. The sim_hal provides a set of API functions used to access the SIM registers including clock gate control and other configuration settings.

Clock Gate Control register access APIs

Clock gate control is based on the module. Each chip has a sub-set of modules that can be gated through gate control registers in SIM. The gate control module names are defined in the fsl_clock_names.h. Each chip also provides a configuration table that defines the supported module names in that chip. Users can access the gate control through the clock manager. See the clock manager for examples to enable a clock module.

Clock Source Control access APIs

Clock source control is also based on the module. Only certain modules have the clock source control in SIM. See the fsl_sim_features.h for available clock sources controlled by the SIM registers. Others are controlled by the device registers. See device drivers for those that are not supported by the SIM registers.

Clock Divider access APIs

Certain clocks use dividers configured in SIM. Only certain dividers are available from SIM. See details in the fsl_sim_features.h for available dividers provided by SIM.

This is an example of SIM HAL access APIs

```
#include "sim/fsl_sim_types.h"
#include "sim/hal/fsl_sim_hal.h"

// calling sim clock gate control API to enable the clock for DMA
SIM_HAL_EnableDmaClock(g_simBaseAddr[0], 0);

#include "sim/fsl_sim_types.h"
#include "sim/hal/fsl_sim_hal.h"

uint8_t setting;

// calling sim clock source control API to get the setting for USBSRC
CLOCK_HAL_GetSource(kClockUsbSrc, &setting);

#include "sim/fsl_sim_types.h"
#include "sim/hal/fsl_sim_hal.h"

uint32_t divider;
```

```
// calling sim divider access API to get the setting for divider outdiv1
CLOCK_HAL_GetDivider(kClockDivideOutdiv1, &divider);
```

33.1.1 Data Structure Documentation

33.1.1.1 struct `clock_name_config_t`

Data Fields

- bool `useOtherRefClock`
if it uses the other ref clock
- `clock_names_t otherRefClockName`
other ref clock name
- `clock_divider_names_t dividerName`
clock divider name

33.1.2 Function Documentation

33.1.2.1 `sim_hal_status_t CLOCK_HAL_SetSource (uint32_t baseAddr, clock_source_names_t clockSource, uint8_t setting)`

This function sets the settings for a specified clock source. Each clock source has its own clock selection settings. See the chip reference manual for clock source detailed settings and the `clock_source_names_t` for clock sources.

Parameters

<code>baseAddr</code>	Base address for current SIM instance.
<code>clockSource</code>	Clock source name defined in <code>sim_clock_source_names_t</code>
<code>setting</code>	Setting value

Returns

`status` If the clock source doesn't exist, it returns an error.

33.1.2.2 `sim_hal_status_t CLOCK_HAL_GetSource (uint32_t baseAddr, clock_source_names_t clockSource, uint8_t * setting)`

This function gets the settings for a specified clock source. Each clock source has its own clock selection settings. See the reference manual for clock source detailed settings and the `clock_source_names_t` for clock sources.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>clockSource</i>	Clock source name
<i>setting</i>	Current setting pointer for the clock source

Returns

status If the clock source doesn't exist, it returns an error.

33.1.2.3 **sim_hal_status_t CLOCK_HAL_SetDivider (uint32_t *baseAddr*, clock_divider_names_t *clockDivider*, uint32_t *setting*)**

This function sets the setting for a specified clock divider. See the reference manual for a supported clock divider and value range and the *clock_divider_names_t* for dividers.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>clockDivider</i>	Clock divider name
<i>setting</i>	Divider setting

Returns

status If the clock divider doesn't exist, it returns an error.

33.1.2.4 **void CLOCK_HAL_SetOutDividers (uint32_t *baseAddr*, uint32_t *outdiv1*, uint32_t *outdiv2*, uint32_t *outdiv3*, uint32_t *outdiv4*)**

This function sets the setting for all clock out dividers at the same time. See the reference manual for a supported clock divider and value range and the *clock_divider_names_t* for clock out dividers.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>outdiv1</i>	Outdivider1 setting

SIM HAL driver

<i>outdiv2</i>	Outdivider2 setting
<i>outdiv3</i>	Outdivider3 setting
<i>outdiv4</i>	Outdivider4 setting

33.1.2.5 **sim_hal_status_t CLOCK_HAL_GetDivider (*uint32_t baseAddr, clock_divider_names_t clockDivider, uint32_t * setting*)**

This function gets the setting for a specified clock divider. See the reference manual for a supported clock divider and value range and the *clock_divider_names_t* for dividers.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>clockDivider</i>	Clock divider name
<i>setting</i>	Divider value pointer

Returns

status If the clock divider doesn't exist, it returns an error.

33.1.2.6 **void SIM_HAL_SetAdcAlternativeTriggerCmd (*uint32_t baseAddr, uint8_t instance, bool enable*)**

This function enables/disables the alternative conversion triggers for ADCx.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.
<i>enable</i>	Enable alternative conversion triggers for ADCx <ul style="list-style-type: none">• true: Select alternative conversion trigger.• false: Select PDB trigger.

33.1.2.7 **bool SIM_HAL_GetAdcAlternativeTriggerCmd (*uint32_t baseAddr, uint8_t instance*)**

This function gets the ADCx alternate trigger enable setting.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.

Returns

enabled True if ADCx alternate trigger is enabled

33.1.2.8 void SIM_HAL_SetAdcPreTriggerMode (uint32_t *baseAddr*, uint8_t *instance*, sim_pretrgsel_t *select*)

This function selects the ADCx pre-trigger source when the alternative triggers are enabled through ADCxALTTRGEN.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.
<i>select</i>	pre-trigger select setting for ADCx <ul style="list-style-type: none"> • 0: Pre-trigger A selected for ADCx. • 1: Pre-trigger B selected for ADCx.

33.1.2.9 sim_pretrgsel_t SIM_HAL_GetAdcPreTriggerMode (uint32_t *baseAddr*, uint8_t *instance*)

This function gets the ADCx pre-trigger select setting.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.

Returns

select ADCx pre-trigger select setting

SIM HAL driver

33.1.2.10 void SIM_HAL_SetAdcTriggerMode (*uint32_t baseAddr, uint8_t instance,* *sim_trgsel_t select*)

This function selects the ADCx trigger source when alternative triggers are enabled through ADCxALT-TRGEN.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.
<i>select</i>	trigger select setting for ADCx <ul style="list-style-type: none"> • 0000: External trigger • 0001: High speed comparator 0 asynchronous interrupt • 0010: High speed comparator 1 asynchronous interrupt • 0011: High speed comparator 2 asynchronous interrupt • 0100: PIT trigger 0 • 0101: PIT trigger 1 • 0110: PIT trigger 2 • 0111: PIT trigger 3 • 1000: FTM0 trigger • 1001: FTM1 trigger • 1010: FTM2 trigger • 1011: FTM3 trigger • 1100: RTC alarm • 1101: RTC seconds • 1110: Low-power timer trigger • 1111: High speed comparator 3 asynchronous interrupt

33.1.2.11 sim_pretrgsel_t SIM_HAL_GetAdcTriggerMode (*uint32_t baseAddr, uint8_t instance*)

This function gets the ADCx trigger select setting.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.

Returns

select ADCx trigger select setting

33.1.2.12 void SIM_HAL_SetUartRxSrcMode (*uint32_t baseAddr, uint8_t instance, sim_uart_rxsrc_t select*)

This function selects the source for the UARTx receive data.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.
<i>select</i>	the source for the UARTx receive data <ul style="list-style-type: none">• 00: UARTx_RX pin.• 01: CMP0.• 10: CMP1.• 11: Reserved.

33.1.2.13 **sim_uart_rxsrc_t SIM_HAL_GetUartRxSrcMode (uint32_t *baseAddr*, uint8_t *instance*)**

This function gets the UARTx receive data source select setting.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.

Returns

select UARTx receive data source select setting

33.1.2.14 **void SIM_HAL_SetUartTxSrcMode (uint32_t *baseAddr*, uint8_t *instance*, sim_uart_txsrc_t *select*)**

This function selects the source for the UARTx transmit data.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.
<i>select</i>	the source for the UARTx transmit data <ul style="list-style-type: none">• 00: UARTx_TX pin.• 01: UARTx_TX pin modulated with FTM1 channel 0 output.• 10: UARTx_TX pin modulated with FTM2 channel 0 output.• 11: Reserved.

33.1.2.15 sim_uart_txsrc_t SIM_HAL_GetUartTxSrcMode (uint32_t *baseAddr*, uint8_t *instance*)

This function gets the UARTx transmit data source select setting.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	device instance.

Returns

select UARTx transmit data source select setting

33.1.2.16 static uint32_t SIM_HAL_GetPinCntId (uint32_t *baseAddr*) [inline], [static]

This function gets the Kinetis Pincount ID in System Device ID register.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
-----------------	--

Returns

id Kinetis Pincount ID

33.1.2.17 static uint32_t SIM_HAL_GetRevId (uint32_t *baseAddr*) [inline], [static]

This function gets the Kinetis Revision ID in System Device ID register.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
-----------------	--

Returns

id Kinetis Revision ID

33.1.2.18 static uint32_t SIM_HAL_GetProgramFlashSize (uint32_t *baseAddr*) [inline], [static]

This function gets the program flash size in the Flash Configuration Register 1.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
-----------------	--

Returns

size Program flash Size

33.1.2.19 void SIM_HAL_EnableDmaClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for DMA moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for DMA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

This function enables the clock for DMA module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.20 void SIM_HAL_DisableDmaClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for DMA moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for DMA module.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.21 **bool SIM_HAL_GetDmaGateCmd (uint32_t *baseAddr*, uint32_t *instance*)**

This function will get the clock gate state for DMA moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for DMA module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.22 **void SIM_HAL_EnableDmamuxClock (uint32_t *baseAddr*, uint32_t *instance*)**

This function enables the clock for DMAMUX moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
-----------------	--

<i>instance</i>	module device instance
-----------------	------------------------

This function enables the clock for DMAMUX module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.23 void SIM_HAL_DisableDmamuxClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for DMAMUX moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for DMAMUX module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.24 bool SIM_HAL_GetDmamuxGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for DMAMUX moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for DMAMUX module.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.25 void SIM_HAL_EnablePortClock (*uint32_t baseAddr, uint32_t instance*)

This function enables the clock for PORT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for PORT module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.26 void SIM_HAL_DisablePortClock (*uint32_t baseAddr, uint32_t instance*)

This function disables the clock for PORT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for PORT module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.27 bool SIM_HAL_GetPortGateCmd (*uint32_t baseAddr, uint32_t instance*)

This function will get the clock gate state for PORT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for PORT module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.28 void SIM_HAL_EnableEwmClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for EWM moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.29 void SIM_HAL_DisableEwmClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for EWM moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.30 bool SIM_HAL_GetEwmGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for EWM moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.31 void SIM_HAL_EnableFtfClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for FTF moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for FTF module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.32 void SIM_HAL_DisableFtfClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for FTF moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for FTF module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.33 bool SIM_HAL_GetFtfGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for FTF moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for FTF module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.34 void SIM_HAL_EnableCrcClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for CRC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.35 void SIM_HAL_DisableCrcClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for CRC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.36 bool SIM_HAL_GetCrcGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for CRC moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.37 void SIM_HAL_EnableAdcClock (*uint32_t baseAddr, uint32_t instance*)

This function enables the clock for ADC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for ADC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.38 void SIM_HAL_DisableAdcClock (*uint32_t baseAddr, uint32_t instance*)

This function disables the clock for ADC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for ADC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.39 bool SIM_HAL_GetAdcGateCmd (*uint32_t baseAddr, uint32_t instance*)

This function will get the clock gate state for ADC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for ADC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.40 void SIM_HAL_EnableCmpClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for CMP moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for CMP module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.41 void SIM_HAL_DisableCmpClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for CMP moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for CMP module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.42 **bool SIM_HAL_GetCmpGateCmd (uint32_t *baseAddr*, uint32_t *instance*)**

This function will get the clock gate state for CMP moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for CMP module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.43 **void SIM_HAL_EnableDacClock (uint32_t *baseAddr*, uint32_t *instance*)**

This function enables the clock for DAC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for DAC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.44 void SIM_HAL_DisableDacClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for DAC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for DAC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.45 bool SIM_HAL_GetDacGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for DAC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for DAC module.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.46 void SIM_HAL_EnableVrefClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for VREF moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.47 void SIM_HAL_DisableVrefClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for VREF moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.48 bool SIM_HAL_GetVrefGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for VREF moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.49 void SIM_HAL_EnableSaiClock (*uint32_t baseAddr, uint32_t instance*)

This function enables the clock for SAI moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.50 void SIM_HAL_DisableSaiClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for SAI moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.51 bool SIM_HAL_GetSaiGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for SAI moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.52 void SIM_HAL_EnablePdbClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for PDB moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.53 void SIM_HAL_DisablePdbClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for PDB moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.54 bool SIM_HAL_GetPdbGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for PDB moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.55 void SIM_HAL_EnableFtmClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for FTM moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.56 void SIM_HAL_DisableFtmClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for FTM moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.57 bool SIM_HAL_GetFtmGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for FTM moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.58 void SIM_HAL_EnablePitClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for PIT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for PIT module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.59 void SIM_HAL_DisablePitClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for PIT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for PIT module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.60 bool SIM_HAL_GetPitGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for PIT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for PIT module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.61 void SIM_HAL_EnableLptimerClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for LPTIMER moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for LPTIMER module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.62 void SIM_HAL_DisableLptimerClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for LPTIMER moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for LPTIMER module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.63 bool SIM_HAL_GetLptimerGateCmd (*uint32_t baseAddr, uint32_t instance*)

This function will get the clock gate state for LPTIMER moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for LPTIMER module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.64 void SIM_HAL_EnableRtcClock (*uint32_t baseAddr, uint32_t instance*)

This function enables the clock for RTC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for RTC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.65 void SIM_HAL_DisableRtcClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for RTC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for RTC module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.66 bool SIM_HAL_GetRtcGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for RTC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for RTC module.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.67 void SIM_HAL_EnableUsbClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for USBFS moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for USBFS module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.68 void SIM_HAL_DisableUsbClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for USBFS moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for USBFS module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.69 bool SIM_HAL_GetUsbGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for USB moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for USB module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.70 void SIM_HAL_EnableSpiClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for SPI moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for SPI module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.71 void SIM_HAL_DisableSpiClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for SPI moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for SPI module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.72 bool SIM_HAL_GetSpiGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for SPI moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for SPI module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.73 void SIM_HAL_EnableI2cClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for I2C moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for I2C module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.74 void SIM_HAL_DisableI2cClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for I2C moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for I2C module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.75 bool SIM_HAL_GetI2cGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for I2C moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for I2C module.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.76 void SIM_HAL_EnableUartClock (*uint32_t baseAddr, uint32_t instance*)

This function enables the clock for UART moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function enables the clock for UART module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.77 void SIM_HAL_DisableUartClock (*uint32_t baseAddr, uint32_t instance*)

This function disables the clock for UART moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

This function disables the clock for UART module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.78 bool SIM_HAL_GetUartGateCmd (*uint32_t baseAddr, uint32_t instance*)

This function will get the clock gate state for UART moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

This function gets the clock gate state for UART module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.79 void SIM_HAL_EnableLpuartClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for LPUART moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.80 void SIM_HAL_DisableLpuartClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for LPUART moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.81 bool SIM_HAL_GetLpuartGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for LPUART moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.82 void SIM_HAL_EnableRngaClock (*uint32_t baseAddr, uint32_t instance*)

This function enables the clock for RNGA moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.83 void SIM_HAL_DisableRngaClock (*uint32_t baseAddr, uint32_t instance*)

This function disables the clock for RNGA moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.84 bool SIM_HAL_GetRngaGateCmd (*uint32_t baseAddr, uint32_t instance*)

This function will get the clock gate state for RNGA moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.85 void SIM_HAL_EnableFlexbusClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for FLEXBUS moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.86 void SIM_HAL_DisableFlexbusClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for FLEXBUS moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.87 bool SIM_HAL_GetFlexbusGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for FLEXBUS moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.88 void SIM_HAL_EnableMpuClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for MPU moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.89 void SIM_HAL_DisableMpuClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for MPU moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.90 **bool SIM_HAL_GetMpuGateCmd (uint32_t *baseAddr*, uint32_t *instance*)**

This function will get the clock gate state for MPU moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.91 **void SIM_HAL_EnableCmtClock (uint32_t *baseAddr*, uint32_t *instance*)**

This function enables the clock for CMT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.92 **void SIM_HAL_DisableCmtClock (uint32_t *baseAddr*, uint32_t *instance*)**

This function disables the clock for CMT moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.93 **bool SIM_HAL_GetCmtGateCmd (uint32_t *baseAddr*, uint32_t *instance*)**

This function will get the clock gate state for CMT moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.94 void SIM_HAL_EnableUsbdcdClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for USBDCD moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.95 void SIM_HAL_DisableUsbdcdClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for USBDCD moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.96 bool SIM_HAL_GetUsbdcdGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for USBDCD moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.97 void SIM_HAL_EnableFlexcanClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for FLEXCAN moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.98 void SIM_HAL_DisableFlexcanClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for FLEXCAN moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.99 bool SIM_HAL_GetFlexcanGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for FLEXCAN moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.100 void SIM_HAL_EnableSdhcClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for SDHC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.101 void SIM_HAL_DisableSdhcClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for SDHC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.102 bool SIM_HAL_GetSdhcGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for SDHC moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.103 void SIM_HAL_EnableEnetClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for ENET moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.104 void SIM_HAL_DisableEnetClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for ENET moudle.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.105 bool SIM_HAL_GetEnetGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function will get the clock gate state for ENET moudle.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.106 void SIM_HAL_EnableTpmClock (uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for TPM module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.107 void SIM_HAL_DisableTpmClock (uint32_t *baseAddr*, uint32_t *instance*)

This function disables the clock for TPM module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.108 bool SIM_HAL_GetTpmGateCmd (uint32_t *baseAddr*, uint32_t *instance*)

This function gets the clock gate state for TPM module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

33.1.2.109 void SIM_HAL_EnableTsiClock(uint32_t *baseAddr*, uint32_t *instance*)

This function enables the clock for TSI module.

SIM HAL driver

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.110 void SIM_HAL_DisableTsiClock (*uint32_t baseAddr, uint32_t instance*)

This function disables the clock for TSI module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

33.1.2.111 bool SIM_HAL_GetTsiGateCmd (*uint32_t baseAddr, uint32_t instance*)

This function gets the clock gate state for TSI module.

Parameters

<i>baseAddr</i>	Base address for current SIM instance.
<i>instance</i>	module device instance

Returns

state true - ungated(Enabled), false - gated (Disabled)

Chapter 34

System Mode Controller (SMC)

The Kinetis SDK provides both HAL and Peripheral drivers for the System Mode Controller (SMC) block of Kinetis devices.

Modules

- [SMC HAL driver](#)

The part describes the programming interface of the SMC HAL driver.

34.1 SMC HAL driver

The chapter describes the programming interface of the SMC HAL driver.

Data Structures

- struct [smc_power_mode_protection_config_t](#)
Power mode protection configuration. [More...](#)
- struct [smc_power_mode_config_t](#)
Power mode control configuration used for calling the SMC_SYS_SetPowerMode API. [More...](#)

Enumerations

- enum [power_modes_t](#)
Power Modes.
- enum [smc_hal_error_code_t](#) {
 kSmcHalSuccess,
 kSmcHalNoSuchModeName,
 kSmcHalAlreadyInTheState,
 kSmcHalFailed }
Error code definition for the system mode controller manager APIs.
- enum [power_mode_stat_t](#) {
 kStatRun = 0x01,
 kStatStop = 0x02,
 kStatVlpr = 0x04,
 kStatVlpw = 0x08,
 kStatVlps = 0x10,
 kStatLls = 0x20,
 kStatVlls = 0x40,
 kStatHsrun = 0x80 }
Power Modes in PMSTAT.
- enum [power_modes_protect_t](#) {
 kAllowHsrun,
 kAllowVlp,
 kAllowLls,
 kAllowVlls }
Power Modes Protection.
- enum [smc_run_mode_t](#) {
 kSmcRun ,
 kSmcVlpr,
 kSmcHsrun }
Run mode definition.
- enum [smc_stop_mode_t](#) {

- kSmcStop,
 - kSmcReservedStop1,
 - kSmcVlps,
 - kSmcLls,
 - kSmcVlls }
- Stop mode definition.*
- enum `smc_stop_submode_t`
- VLLS/LLS stop sub mode definition.*
- enum `smc_lpwui_option_t` {
- `kSmcLpwuiEnabled,`
 - `kSmcLpwuiDisabled }`
- Low Power Wake Up on Interrupt option.*
- enum `smc_pstop_option_t` {
- `kSmcPstopStop,`
 - `kSmcPstopStop1,`
 - `kSmcPstopStop2 }`
- Partial STOP option.*
- enum `smc_por_option_t` {
- `kSmcPorEnabled,`
 - `kSmcPorDisabled }`
- POR option.*
- enum `smc_lpo_option_t` {
- `kSmcLpoEnabled,`
 - `kSmcLpoDisabled }`
- LPO power option.*
- enum `smc_power_options_t` {
- `kSmcOptionLpwui,`
 - `kSmcOptionPropo }`
- Power mode control options.*

System mode controller APIs

- `smc_hal_error_code_t SMC_HAL_SetMode` (`uint32_t baseAddr, const smc_power_mode_config_t *powerModeConfig)`
Configures the power mode.
- `void SMC_HAL_SetProtection` (`uint32_t baseAddr, smc_power_mode_protection_config_t *protectConfig)`
Configures all power mode protection settings.
- `void SMC_HAL_SetProtectionMode` (`uint32_t baseAddr, power_modes_protect_t protect, bool allow)`
Configures the individual power mode protection settings.
- `bool SMC_HAL_GetProtectionMode` (`uint32_t baseAddr, power_modes_protect_t protect)`
Gets the the current power mode protection setting.
- `void SMC_HAL_SetRunMode` (`uint32_t baseAddr, smc_run_mode_t runMode)`
Configures the the RUN mode control setting.
- `smc_run_mode_t SMC_HAL_GetRunMode` (`uint32_t baseAddr)`
Gets the current RUN mode configuration setting.

SMC HAL driver

- void **SMC_HAL_SetStopMode** (uint32_t baseAddr, smc_stop_mode_t stopMode)
Configures the STOP mode control setting.
- smc_stop_mode_t **SMC_HAL_GetStopMode** (uint32_t baseAddr)
Gets the current STOP mode control settings.
- void **SMC_HAL_SetStopSubMode** (uint32_t baseAddr, smc_stop_submode_t stopSubMode)
Configures the stop sub mode control setting.
- smc_stop_submode_t **SMC_HAL_GetStopSubMode** (uint32_t baseAddr)
Gets the current stop submode configuration settings.
- uint8_t **SMC_HAL_GetStat** (uint32_t baseAddr)
Gets the current power mode stat.

34.1.0.112 SMC Hal Driver

Overview

The System Mode Controller (SMC) sequences the system in and out of all low-power stop and run modes. Specifically, it monitors events to trigger transitions between the power modes while controlling the power, clocks, and memories of the system to achieve the power consumption and functionality of that mode. It also provides a set of functions to configure the power mode protection, the power mode, and other configuration settings.

Power Mode Configuration APIs

- Power Mode Protection configurations APIs Allow the Very Low Power mode / Read the current mode Allow the Low Leakage Stop mode / Read the current mode Allow the Very Low Leakage Stop mode / Read the current mode
- Power Mode Control configurations APIs
- VLLS Mode Control configurations APIs
- Power Mode Status APIs

This is an example of the SMC manager APIs.

```
#include "fsl_smcc_hal.h"

/* power mode config structure */
smc_power_mode_config_t smcConfig;

/* power mode and option mode */
smcConfig.lpwuiOption = false;
smcConfig.porOption = false;
smcConfig.powerModeName = kPowerModeRun;

/* set the power mode */
SMC_HAL_SetMode(&smcConfig);
```

Control register access APIs

- Power mode configurations register access

- VLLS mode configurations register access
- Power Mode Status access

This is an example of the SMC HAL access APIs.

```
#include "fsl_smcc_hal.h"

/* protection mode */
smc_power_mode_protection_config_t smcProtConfig;

/* set to allow entering specific protect mode */
smcProtConfig.llsProt = true;
smcProtConfig.vllsProt = true;
smcProtConfig.vlpProt = true;

/* set protection modes */
SMC_HAL_SetProtection(&smcProtConfig);
```

34.1.1 Data Structure Documentation

34.1.1.1 struct smc_power_mode_protection_config_t

Data Fields

- bool **vlpProt**
VLP protect.
- bool **llsProt**
LLS protect.
- bool **vllsProt**
VLLS protect.

34.1.1.2 struct smc_power_mode_config_t

Data Fields

- **power_modes_t powerModeName**
Power mode(enum), see power_modes_t.
- **smc_stop_submode_t stopSubMode**
Stop submode(enum), see smc_stop_submode_t.

34.1.2 Enumeration Type Documentation

34.1.2.1 enum smc_hal_error_code_t

Enumerator

kSmcHalSuccess Success.

kSmcHalNoSuchModeName Cannot find the mode name specified.

kSmcHalAlreadyInTheState Already in the required state.

SMC HAL driver

kSmcHalFailed Unknown error, operation failed.

34.1.2.2 enum power_mode_stat_t

Enumerator

kStatRun 0000_0001 - Current power mode is RUN
kStatStop 0000_0010 - Current power mode is STOP
kStatVlpr 0000_0100 - Current power mode is VLPR
kStatVlpw 0000_1000 - Current power mode is VLPW
kStatVlps 0001_0000 - Current power mode is VLPS
kStatLls 0010_0000 - Current power mode is LLS
kStatVlls 0100_0000 - Current power mode is VLLS
kStatHsrun 1000_0000 - Current power mode is HSRUN

34.1.2.3 enum power_modes_protect_t

Enumerator

kAllowHsrun Allow High Speed Run mode.
kAllowVlp Allow Very-Low-Power Modes.
kAllowLls Allow Low-Leakage Stop Mode.
kAllowVlls Allow Very-Low-Leakage Stop Mode.

34.1.2.4 enum smc_run_mode_t

Enumerator

kSmcRun normal RUN mode
kSmcVlpr Very-Low-Power RUN mode.
kSmcHsrun High Speed Run mode (HSRUN)

34.1.2.5 enum smc_stop_mode_t

Enumerator

kSmcStop Normal STOP mode.
kSmcReservedStop1 Reserved.
kSmcVlps Very-Low-Power STOP mode.
kSmcLls Low-Leakage Stop mode.
kSmcVlls Very-Low-Leakage Stop mode.

34.1.2.6 enum smc_lpwui_option_t

Enumerator

kSmcLpwuiEnabled Low Power Wake Up on Interrupt enabled.

kSmcLpwuiDisabled Low Power Wake Up on Interrupt disabled.

34.1.2.7 enum smc_pstop_option_t

Enumerator

kSmcPstopStop STOP - Normal Stop mode.

kSmcPstopStop1 Partial Stop with both system and bus clocks disabled.

kSmcPstopStop2 Partial Stop with system clock disabled and bus clock enabled.

34.1.2.8 enum smc_por_option_t

Enumerator

kSmcPorEnabled POR detect circuit is enabled in VLLS0.

kSmcPorDisabled POR detect circuit is disabled in VLLS0.

34.1.2.9 enum smc_lpo_option_t

Enumerator

kSmcLpoEnabled LPO clock is enabled in LLS/VLLSx.

kSmcLpoDisabled LPO clock is disabled in LLS/VLLSx.

34.1.2.10 enum smc_power_options_t

Enumerator

kSmcOptionLpwui Low Power Wake Up on Interrupt.

kSmcOptionPropo POR option.

34.1.3 Function Documentation

34.1.3.1 smc_hal_error_code_t SMC_HAL_SetMode (uint32_t baseAddr, const smc_power_mode_config_t * powerModeConfig)

This function configures the power mode control for both run, stop, and stop sub mode if needed. Also it configures the power options for a specific power mode. An application should follow the proper procedure

SMC HAL driver

to configure and switch power modes between different run and stop modes. For proper procedures and supported power modes, see an appropriate chip reference manual. See the [smc_power_mode_config_t](#) for required parameters to configure the power mode and the supported options. Other options may need to be individually configured through the HAL driver. See the HAL driver header file for details.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>powerModeConfig</i>	Power mode configuration structure smc_power_mode_config_t

Returns

errorCode SMC error code

34.1.3.2 void SMC_HAL_SetProtection (*uint32_t baseAddr*, *smc_power_mode_protection_config_t * protectConfig*)

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the [smc_power_mode_protection_config_t](#). An application should provide the protect settings for all supported power modes on the chip. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset. If the user has only a single option to set, either use this function or use the individual set function.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>protectConfig</i>	Configurations for the supported power mode protect settings <ul style="list-style-type: none">• See smc_power_mode_protection_config_t for details.

34.1.3.3 void SMC_HAL_SetProtectionMode (*uint32_t baseAddr*, *power_modes_protect_t protect*, *bool allow*)

This function only configures the power mode protection settings for a specified power mode on the specified chip family. The available power modes are defined in the [smc_power_mode_protection_config_t](#). See the reference manual for details. This register can only write once after the power reset.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>protect</i>	Power mode to set for protection
<i>allow</i>	Allow or not allow the power mode protection

34.1.3.4 **bool SMC_HAL_GetProtectionMode (uint32_t *baseAddr*, power_modes_protect_t *protect*)**

This function gets the current power mode protection settings for a specified power mode.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>protect</i>	Power mode to set for protection

Returns

state Status of the protection setting

- true: Allowed
- false: Not allowed

34.1.3.5 **void SMC_HAL_SetRunMode (uint32_t *baseAddr*, smc_run_mode_t *runMode*)**

This function sets the run mode settings, for example, normal run mode, very lower power run mode, etc. See the *smc_run_mode_t* for supported run mode on the chip family and the reference manual for details about the run mode.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>runMode</i>	Run mode setting defined in <i>smc_run_mode_t</i>

34.1.3.6 **smc_run_mode_t SMC_HAL_GetRunMode (uint32_t *baseAddr*)**

This function gets the run mode settings. See the *smc_run_mode_t* for a supported run mode on the chip family and the reference manual for details about the run mode.

SMC HAL driver

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
-----------------	--

Returns

setting Run mode configuration setting

34.1.3.7 void SMC_HAL_SetStopMode (uint32_t *baseAddr*, smc_stop_mode_t *stopMode*)

This function sets the stop mode settings, for example, normal stop mode, very lower power stop mode, etc. See the smc_stop_mode_t for supported stop mode on the chip family and the reference manual for details about the stop mode.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>stopMode</i>	Stop mode defined in smc_stop_mode_t

34.1.3.8 smc_stop_mode_t SMC_HAL_GetStopMode (uint32_t *baseAddr*)

This function gets the stop mode settings, for example, normal stop mode, very lower power stop mode, etc. See the smc_stop_mode_t for supported stop mode on the chip family and the reference manual for details about the stop mode.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
-----------------	--

Returns

setting Current stop mode configuration setting

34.1.3.9 void SMC_HAL_SetStopSubMode (uint32_t *baseAddr*, smc_stop_submode_t *stopSubMode*)

This function sets the stop submode settings. Some of the stop mode further supports submodes. See the smc_stop_submode_t for supported stop submodes and the reference manual for details about the submodes for a specific stop mode.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
<i>stopSubMode</i>	Stop submode setting defined in smc_stop_submode_t

34.1.3.10 smc_stop_submode_t SMC_HAL_GetStopSubMode (uint32_t *baseAddr*)

This function gets the stop submode settings. Some of the stop mode further support submodes. See the smc_stop_submode_t for supported stop submodes and the reference manual for details about the submode for a specific stop mode.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
-----------------	--

Returns

setting Current stop submode setting

34.1.3.11 uint8_t SMC_HAL_GetStat (uint32_t *baseAddr*)

This function returns the current power mode stat. Once application switches the power mode, it should always check the stat to check whether it runs into the specified mode or not. An application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the _power_mode_stat for information about the power stat.

Parameters

<i>baseAddr</i>	Base address for current SMC instance.
-----------------	--

Returns

stat Current power mode stat

Chapter 35

Clock Manager (Clock)

The Kinetis SDK Clock Manager provides a set of API/services to configure the clock-related IPs, such as MCG, SIM, etc.

Enumerations

- enum `clock_manager_error_code_t` {
 `kClockManagerSuccess`,
 `kClockManagerNoSuchClockName`,
 `kClockManagerNoSuchClockModule`,
 `kClockManagerNoSuchClockSource`,
 `kClockManagerNoSuchDivider`,
 `kClockManagerUnknown` }

Error code definition for the clock manager APIs.

Functions

- `uint32_t CLOCK_SYS_GetDmaFreq (uint32_t instance)`
Gets the clock frequency for DMA module.
- `uint32_t CLOCK_SYS_GetDmamuxFreq (uint32_t instance)`
Gets the clock frequency for DMAMUX module.
- `uint32_t CLOCK_SYS_GetPortFreq (uint32_t instance)`
Gets the clock frequency for PORT module.
- `uint32_t CLOCK_SYS_GetEwmFreq (uint32_t instance)`
Gets the clock frequency for EWM module.
- `uint32_t CLOCK_SYS_GetFtfFreq (uint32_t instance)`
Gets the clock frequency for FTF module.
- `uint32_t CLOCK_SYS_GetCrcFreq (uint32_t instance)`
Gets the clock frequency for CRC module.
- `uint32_t CLOCK_SYS_GetAdcFreq (uint32_t instance)`
Gets the clock frequency for ADC module.
- `uint32_t CLOCK_SYS_GetCmpFreq (uint32_t instance)`
Gets the clock frequency for CMP module.
- `uint32_t CLOCK_SYS_GetVrefFreq (uint32_t instance)`
Gets the clock frequency for VREF module.
- `uint32_t CLOCK_SYS_GetPdbFreq (uint32_t instance)`
Gets the clock frequency for PDB module.
- `uint32_t CLOCK_SYS_GetFtmFreq (uint32_t instance)`
Gets the clock frequency for FTM module.
- `uint32_t CLOCK_SYS_GetPitFreq (uint32_t instance)`
Gets the clock frequency for PIT module.
- `uint32_t CLOCK_SYS_GetUsbFreq (uint32_t instance)`
Gets the clock frequency for USB FS OTG module.
- `uint32_t CLOCK_SYS_GetSpiFreq (uint32_t instance)`

- `uint32_t CLOCK_SYS_GetI2cFreq (uint32_t instance)`
Gets the clock frequency for I2C module.
- `uint32_t CLOCK_SYS_GetUartFreq (uint32_t instance)`
Gets the clock frequency for UART module.
- `uint32_t CLOCK_SYS_GetLpuartFreq (uint32_t instance)`
Gets the clock frequency for LPUART module.
- `uint32_t CLOCK_SYS_GetSaiFreq (uint32_t instance)`
Gets the clock frequency for I2S module.
- `uint32_t CLOCK_SYS_GetGpioFreq (uint32_t instance)`
Gets the clock frequency for GPIO module.
- `static void CLOCK_SYS_EnableDmaClock (uint32_t instance)`
Enable the clock for DMA module.
- `static void CLOCK_SYS_DisableDmaClock (uint32_t instance)`
Disable the clock for DMA module.
- `static bool CLOCK_SYS_GetDmaGateCmd (uint32_t instance)`
Get the the clock gate state for DMA module.
- `static void CLOCK_SYS_EnableDmamuxClock (uint32_t instance)`
Enable the clock for DMAMUX module.
- `static void CLOCK_SYS_DisableDmamuxClock (uint32_t instance)`
Disable the clock for DMAMUX module.
- `static bool CLOCK_SYS_GetDmamuxGateCmd (uint32_t instance)`
Get the the clock gate state for DMAMUX module.
- `static void CLOCK_SYS_EnablePortClock (uint32_t instance)`
Enable the clock for PORT module.
- `static void CLOCK_SYS_DisablePortClock (uint32_t instance)`
Disable the clock for PORT module.
- `static bool CLOCK_SYS_GetPortGateCmd (uint32_t instance)`
Get the the clock gate state for PORT module.
- `static void CLOCK_SYS_EnableEwmClock (uint32_t instance)`
Enable the clock for EWM module.
- `static void CLOCK_SYS_DisableEwmClock (uint32_t instance)`
Disable the clock for EWM module.
- `static bool CLOCK_SYS_GetEwmGateCmd (uint32_t instance)`
Get the the clock gate state for EWM module.
- `static void CLOCK_SYS_EnableFtfClock (uint32_t instance)`
Enable the clock for FTF module.
- `static void CLOCK_SYS_DisableFtfClock (uint32_t instance)`
Disable the clock for FTF module.
- `static bool CLOCK_SYS_GetFtfGateCmd (uint32_t instance)`
Get the the clock gate state for FTF module.
- `static void CLOCK_SYS_EnableCrcClock (uint32_t instance)`
Enable the clock for CRC module.
- `static void CLOCK_SYS_DisableCrcClock (uint32_t instance)`
Disable the clock for CRC module.
- `static bool CLOCK_SYS_GetCrcGateCmd (uint32_t instance)`
Get the the clock gate state for CRC module.
- `static void CLOCK_SYS_EnableAdcClock (uint32_t instance)`
Enable the clock for ADC module.
- `static void CLOCK_SYS_DisableAdcClock (uint32_t instance)`
Disable the clock for ADC module.

- static bool **CLOCK_SYS_GetAdcGateCmd** (uint32_t instance)
Get the the clock gate state for ADC module.
- static void **CLOCK_SYS_EnableCmpClock** (uint32_t instance)
Enable the clock for CMP module.
- static void **CLOCK_SYS_DisableCmpClock** (uint32_t instance)
Disable the clock for CMP module.
- static bool **CLOCK_SYS_GetCmpGateCmd** (uint32_t instance)
Get the the clock gate state for CMP module.
- static void **CLOCK_SYS_EnableDacClock** (uint32_t instance)
Enable the clock for DAC module.
- static void **CLOCK_SYS_DisableDacClock** (uint32_t instance)
Disable the clock for DAC module.
- static bool **CLOCK_SYS_GetDacGateCmd** (uint32_t instance)
Get the the clock gate state for DAC module.
- static void **CLOCK_SYS_EnableVrefClock** (uint32_t instance)
Enable the clock for VREF module.
- static void **CLOCK_SYS_DisableVrefClock** (uint32_t instance)
Disable the clock for VREF module.
- static bool **CLOCK_SYS_GetVrefGateCmd** (uint32_t instance)
Get the the clock gate state for VREF module.
- static void **CLOCK_SYS_EnableSaiClock** (uint32_t instance)
Enable the clock for SAI module.
- static void **CLOCK_SYS_DisableSaiClock** (uint32_t instance)
Disable the clock for SAI module.
- static bool **CLOCK_SYS_GetSaiGateCmd** (uint32_t instance)
Get the the clock gate state for SAI module.
- static void **CLOCK_SYS_EnablePdbClock** (uint32_t instance)
Enable the clock for PDB module.
- static void **CLOCK_SYS_DisablePdbClock** (uint32_t instance)
Disable the clock for PDB module.
- static bool **CLOCK_SYS_GetPdbGateCmd** (uint32_t instance)
Get the the clock gate state for PDB module.
- static void **CLOCK_SYS_EnableFtmClock** (uint32_t instance)
Enable the clock for FTM module.
- static void **CLOCK_SYS_DisableFtmClock** (uint32_t instance)
Disable the clock for FTM module.
- static bool **CLOCK_SYS_GetFtmGateCmd** (uint32_t instance)
Get the the clock gate state for FTM module.
- static void **CLOCK_SYS_EnablePitClock** (uint32_t instance)
Enable the clock for PIT module.
- static void **CLOCK_SYS_DisablePitClock** (uint32_t instance)
Disable the clock for PIT module.
- static bool **CLOCK_SYS_GetPitGateCmd** (uint32_t instance)
Get the the clock gate state for PIT module.
- static void **CLOCK_SYS_EnableLptimerClock** (uint32_t instance)
Enable the clock for LPTIMER module.
- static void **CLOCK_SYS_DisableLptimerClock** (uint32_t instance)
Disable the clock for LPTIMER module.
- static bool **CLOCK_SYS_GetLptimerGateCmd** (uint32_t instance)
Get the the clock gate state for LPTIMER module.
- static void **CLOCK_SYS_EnableRtcClock** (uint32_t instance)

- static void **CLOCK_SYS_DisableRtcClock** (uint32_t instance)

Disable the clock for RTC module.
- static bool **CLOCK_SYS_GetRtcGateCmd** (uint32_t instance)

Get the the clock gate state for RTC module.
- static void **CLOCK_SYS_EnableUsbClock** (uint32_t instance)

Enable the clock for USBFS module.
- static void **CLOCK_SYS_DisableUsbClock** (uint32_t instance)

Disable the clock for USBFS module.
- static bool **CLOCK_SYS_GetUsbGateCmd** (uint32_t instance)

Get the the clock gate state for USB module.
- static void **CLOCK_SYS_EnableSpiClock** (uint32_t instance)

Enable the clock for SPI module.
- static void **CLOCK_SYS_DisableSpiClock** (uint32_t instance)

Disable the clock for SPI module.
- static bool **CLOCK_SYS_GetSpiGateCmd** (uint32_t instance)

Get the the clock gate state for SPI module.
- static void **CLOCK_SYS_EnableI2cClock** (uint32_t instance)

Enable the clock for I2C module.
- static void **CLOCK_SYS_DisableI2cClock** (uint32_t instance)

Disable the clock for I2C module.
- static bool **CLOCK_SYS_GetI2cGateCmd** (uint32_t instance)

Get the the clock gate state for I2C module.
- static void **CLOCK_SYS_EnableUartClock** (uint32_t instance)

Enable the clock for UART module.
- static void **CLOCK_SYS_DisableUartClock** (uint32_t instance)

Disable the clock for UART module.
- static bool **CLOCK_SYS_GetUartGateCmd** (uint32_t instance)

Get the the clock gate state for UART module.
- static void **CLOCK_SYS_EnableLpuartClock** (uint32_t instance)

Enable the clock for LPUART module.
- static void **CLOCK_SYS_DisableLpuartClock** (uint32_t instance)

Disable the clock for LPUART module.
- static bool **CLOCK_SYS_GetLpuartGateCmd** (uint32_t instance)

Get the the clock gate state for LPUART module.
- uint32_t **CLOCK_SYS_GetRngaFreq** (uint32_t instance)

Gets the clock frequency for RNGA module.
- static void **CLOCK_SYS_EnableRngaClock** (uint32_t instance)

Enable the clock for RNGA module.
- static void **CLOCK_SYS_DisableRngaClock** (uint32_t instance)

Disable the clock for RNGA module.
- static bool **CLOCK_SYS_GetRngaGateCmd** (uint32_t instance)

Get the the clock gate state for RNGA module.
- uint32_t **CLOCK_SYS_GetFlexbusFreq** (uint32_t instance)

Gets the clock frequency for FLEXBUS module.
- static void **CLOCK_SYS_EnableFlexbusClock** (uint32_t instance)

Enable the clock for FLEXBUS module.
- static void **CLOCK_SYS_DisableFlexbusClock** (uint32_t instance)

Disable the clock for FLEXBUS module.
- static bool **CLOCK_SYS_GetFlexbusGateCmd** (uint32_t instance)

Get the the clock gate state for FLEXBUS module.

- `uint32_t CLOCK_SYS_GetMpuFreq (uint32_t instance)`
Gets the clock frequency for MPU module.
- `uint32_t CLOCK_SYS_GetCmtFreq (uint32_t instance)`
Gets the clock frequency for CMT module.
- `uint32_t CLOCK_SYS_GetEnetRmiiFreq (uint32_t instance)`
Gets the clock frequency for ENET module RMII clock.
- `uint32_t CLOCK_SYS_GetEnetTimeStampFreq (uint32_t instance)`
Gets the clock frequency for ENET module TIME clock.
- `uint32_t CLOCK_SYS_GetUsbdcdFreq (uint32_t instance)`
Gets the clock frequency for USB DCD module.
- `uint32_t CLOCK_SYS_GetSdhcFreq (uint32_t instance)`
Gets the clock frequency for SDHC module.
- `static void CLOCK_SYS_EnableMpuClock (uint32_t instance)`
Enable the clock for MPU module.
- `static void CLOCK_SYS_DisableMpuClock (uint32_t instance)`
Disable the clock for MPU module.
- `static bool CLOCK_SYS_GetMpuGateCmd (uint32_t instance)`
Get the the clock gate state for MPU module.
- `static void CLOCK_SYS_EnableCmtClock (uint32_t instance)`
Enable the clock for CMT module.
- `static void CLOCK_SYS_DisableCmtClock (uint32_t instance)`
Disable the clock for CMT module.
- `static bool CLOCK_SYS_GetCmtGateCmd (uint32_t instance)`
Get the the clock gate state for CMT module.
- `static void CLOCK_SYS_EnableUsbdcdClock (uint32_t instance)`
Enable the clock for USBD_CD module.
- `static void CLOCK_SYS_DisableUsbdcdClock (uint32_t instance)`
Disable the clock for USBD_CD module.
- `static bool CLOCK_SYS_GetUsbdcdGateCmd (uint32_t instance)`
Get the the clock gate state for USBD_CD module.
- `static void CLOCK_SYS_EnableFlexcanClock (uint32_t instance)`
Enable the clock for FLEXCAN module.
- `static void CLOCK_SYS_DisableFlexcanClock (uint32_t instance)`
Disable the clock for FLEXCAN module.
- `static bool CLOCK_SYS_GetFlexcanGateCmd (uint32_t instance)`
Get the the clock gate state for FLEXCAN module.
- `static void CLOCK_SYS_EnableSdhcClock (uint32_t instance)`
Enable the clock for SDHC module.
- `static void CLOCK_SYS_DisableSdhcClock (uint32_t instance)`
Disable the clock for SDHC module.
- `static bool CLOCK_SYS_GetSdhcGateCmd (uint32_t instance)`
Get the the clock gate state for SDHC module.
- `static void CLOCK_SYS_EnableEnetClock (uint32_t instance)`
Enable the clock for ENET module.
- `static void CLOCK_SYS_DisableEnetClock (uint32_t instance)`
Disable the clock for ENET module.
- `static bool CLOCK_SYS_GetEnetGateCmd (uint32_t instance)`
Get the the clock gate state for ENET module.
- `uint32_t CLOCK_SYS_GetTpmFreq (uint32_t instance)`
Gets the clock frequency for TPM module.
- `static void CLOCK_SYS_EnableTpmClock (uint32_t instance)`

- static void **CLOCK_SYS_DisableTpmClock** (uint32_t instance)

Disable the clock for TPM module.
- static bool **CLOCK_SYS_GetTpmGateCmd** (uint32_t instance)

Get the the clock gate state for TPM module.
- static void **CLOCK_SYS_EnableTsiClock** (uint32_t instance)

Enable the clock for TSI module.
- static void **CLOCK_SYS_DisableTsiClock** (uint32_t instance)

Disable the clock for TSI module.
- static bool **CLOCK_SYS_GetTsiGateCmd** (uint32_t instance)

Get the the clock gate state for TSI module.

Clock Frequencies

- **clock_manager_error_code_t CLOCK_SYS_GetFreq** (clock_names_t clockName, uint32_t *frequency)

Gets the clock frequency for a specific clock name.
- **clock_manager_error_code_t CLOCK_SYS_SetSource** (clock_source_names_t clockSource, uint8_t setting)

Sets the clock source setting.
- **clock_manager_error_code_t CLOCK_SYS_GetSource** (clock_source_names_t clockSource, uint8_t *setting)

Gets the clock source setting.
- **clock_manager_error_code_t CLOCK_SYS_SetDivider** (clock_divider_names_t clockDivider, uint32_t setting)

Sets the clock divider setting.
- **clock_manager_error_code_t CLOCK_SYS_GetDivider** (clock_divider_names_t clockDivider, uint32_t *setting)

Gets the clock divider setting.
- static void **CLOCK_SYS_SetOutDividers** (uint32_t outdiv1, uint32_t outdiv2, uint32_t outdiv3, uint32_t outdiv4)

Sets the clock out dividers setting.

35.0.4 Clock Manager

Overview

The Clock Manager is configures, accesses core, platform, system and bus clock setting. It provides a set of APIs to get and set functions of the clock gate control.

It includes manager-level, SIM HAL-level, and MCG HAL-level access APIs.

Clock names, Clock source names and Clock module names

There are three sets of clock related names: clock names, clock source names, and clock module names. Each CPU only supports a subset of clock related names. If there is a new clock or a clock module that does exist in the current name set, it has to be added in the definition.

Clock names get a system clock frequency. The clock module names control and access the clock module gate status for a specific module.

Example to get a system clock frequency:

```
#include "clock/fsl_clock_manager.h"
uint32_t uFrequency = 0;

// get the system clock frequency based on current mode configuration
if (kClockManagerSuccess == CLOCK_SYS_GetFreq(kSystemClock, &frequency
    ))
    // check the frequency value for system clock in frequency
```

Example to get a clock frequency based on the clock source:

```
#include "fsl_clock_manager.h"
uint32_t frequency = 0;

// get the UART clock frequency based on current mode configuration
frequency = CLOCK_SYS_GetUartFreq(instance);
```

Example to enable a clock module:

```
#include "fsl_sim_hal.h"

// enable the ENET clock
SIM_HAL_EnableEnetClock(g_simBaseAddr[0], 0);
```

35.1 Enumeration Type Documentation

35.1.1 enum clock_manager_error_code_t

Enumerator

- kClockManagerSuccess* success
- kClockManagerNoSuchClockName* cannot find the clock name
- kClockManagerNoSuchClockModule* cannot find the clock module name
- kClockManagerNoSuchClockSource* cannot find the clock source name
- kClockManagerNoSuchDivider* cannot find the divider name
- kClockManagerUnknown* unknown error

35.2 Function Documentation

35.2.1 `clock_manager_error_code_t CLOCK_SYS_GetFreq (clock_names_t clockName, uint32_t * frequency)`

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in `clock_names_t`. The MCG must be properly configured before using this function. See the reference manual for supported clock names for different chip families. The returned value is in Hertz. If it cannot find the clock name or the name is not supported for a specific chip family, it returns an error.

Function Documentation

Parameters

<i>clockName</i>	Clock names defined in <code>clock_names_t</code>
<i>frequency</i>	Returned clock frequency value in Hertz

Returns

`status` Error code defined in `clock_manager_error_code_t`

35.2.2 `clock_manager_error_code_t CLOCK_SYS_SetSource (clock_source_names_t clockSource, uint8_t setting)`

This function sets the settings for a specified clock source. Each clock source has its own clock selection settings. See the chip reference manual for clock source detailed settings and the `sim_clock_source_names_t` for clock sources.

Parameters

<i>clockSource</i>	Clock source name defined in <code>sim_clock_source_names_t</code>
<i>setting</i>	Setting value

Returns

`status` If the clock source doesn't exist, it returns an error.

35.2.3 `clock_manager_error_code_t CLOCK_SYS_GetSource (clock_source_names_t clockSource, uint8_t * setting)`

This function gets the settings for a specified clock source. Each clock source has its own clock selection settings. See the reference manual for clock source detailed settings and the `sim_clock_source_names_t` for clock sources.

Parameters

<i>clockSource</i>	Clock source name
--------------------	-------------------

<i>setting</i>	Current setting for the clock source
----------------	--------------------------------------

Returns

status If the clock source doesn't exist, it returns an error.

35.2.4 **clock_manager_error_code_t CLOCK_SYS_SetDivider (clock_divider_names_t *clockDivider*, uint32_t *setting*)**

This function sets the setting for a specified clock divider. See the reference manual for a supported clock divider and value range and the sim_clock_divider_names_t for dividers.

Parameters

<i>clockDivider</i>	Clock divider name
<i>divider</i>	Divider setting

Returns

status If the clock divider doesn't exist, it returns an error.

35.2.5 **clock_manager_error_code_t CLOCK_SYS_GetDivider (clock_divider_names_t *clockDivider*, uint32_t * *setting*)**

This function gets the setting for a specified clock divider. See the reference manual for a supported clock divider and value range and the clock_divider_names_t for dividers.

Parameters

<i>clockDivider</i>	Clock divider name
<i>divider</i>	Divider value pointer

Returns

status If the clock divider doesn't exist, it returns an error.

35.2.6 **static void CLOCK_SYS_SetOutDividers (uint32_t *outdiv1*, uint32_t *outdiv2*, uint32_t *outdiv3*, uint32_t *outdiv4*) [inline], [static]**

This function sets the setting for all clock out dividers at the same time. See the reference manual for a supported clock divider and value range and the clock_divider_names_t for clock out dividers.

Function Documentation

Parameters

<i>outdiv1</i>	Outdivider1 setting
<i>outdiv2</i>	Outdivider2 setting
<i>outdiv3</i>	Outdivider3 setting
<i>outdiv4</i>	Outdivider4 setting

35.2.7 **uint32_t CLOCK_SYS_GetDmaFreq (uint32_t *instance*)**

This function gets the clock frequency for DMA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for DMA module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.8 **uint32_t CLOCK_SYS_GetDmamuxFreq (uint32_t *instance*)**

This function gets the clock frequency for DMAMUX moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for DMAMUX module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.9 **uint32_t CLOCK_SYS_GetPortFreq (uint32_t *instance*)**

This function gets the clock frequency for PORT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for PORT module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.10 **uint32_t CLOCK_SYS_GetEwmFreq (uint32_t *instance*)**

This function gets the clock frequency for EWM moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for EWM module.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.11 uint32_t CLOCK_SYS_GetFtfFreq (uint32_t *instance*)

(Flash Memory)

This function gets the clock frequency for FTF module. (Flash Memory)

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

(Flash Memory)

This function gets the clock frequency for FTF module. (Flash Memory)

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.12 uint32_t CLOCK_SYS_GetCrcFreq (uint32_t *instance*)

This function gets the clock frequency for CRC module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for CRC module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.13 uint32_t CLOCK_SYS_GetAdcFreq (uint32_t *instance*)

This function gets the clock frequency for ADC module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for ADC module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.14 uint32_t CLOCK_SYS_GetCmpFreq (uint32_t *instance*)

This function gets the clock frequency for CMP module.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for CMP module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.15 uint32_t CLOCK_SYS_GetVrefFreq (uint32_t *instance*)

This function gets the clock frequency for VREF module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for VREF module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.16 uint32_t CLOCK_SYS_GetPdbFreq (uint32_t *instance*)

This function gets the clock frequency for PDB module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for PDB module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.17 **uint32_t CLOCK_SYS_GetFtmFreq (uint32_t *instance*)**

(FlexTimer)

This function gets the clock frequency for FTM module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

(FlexTimer)

This function gets the clock frequency for FTM module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.18 **uint32_t CLOCK_SYS_GetPitFreq (uint32_t *instance*)**

This function gets the clock frequency for PIT module.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for PIT module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.19 `uint32_t CLOCK_SYS_GetUsbFreq (uint32_t instance)`

This function gets the clock frequency for USB FS OTG module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for USB FS OTG module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.20 `uint32_t CLOCK_SYS_GetSpiFreq (uint32_t instance)`

This function gets the clock frequency for SPI module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for SPI module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.21 **uint32_t CLOCK_SYS_GetI2cFreq (uint32_t *instance*)**

This function gets the clock frequency for I2C module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for I2C module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.22 **uint32_t CLOCK_SYS_GetUartFreq (uint32_t *instance*)**

This function gets the clock frequency for UART module.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for UART module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.23 uint32_t CLOCK_SYS_GetLpuartFreq (uint32_t *instance*)

This function gets the clock frequency for LPUART module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for LPUART module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.24 uint32_t CLOCK_SYS_GetSaiFreq (uint32_t *instance*)

This function gets the clock frequency for I2S module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for I2S module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.25 uint32_t CLOCK_SYS_GetGpioFreq (uint32_t *instance*)

This function gets the clock frequency for GPIO module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for GPIO module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.26 static void CLOCK_SYS_EnableDmaClock (uint32_t *instance*) [inline], [static]

This function enables the clock for DMA moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.27 static void CLOCK_SYS_DisableDmaClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for DMA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.28 static bool CLOCK_SYS_GetDmaGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for DMA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.29 static void CLOCK_SYS_EnableDmamuxClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for DMAMUX moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.30 static void CLOCK_SYS_DisableDmamuxClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for DMAMUX moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.31 static bool CLOCK_SYS_GetDmamuxGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for DMAMUX moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.32 static void CLOCK_SYS_EnablePortClock (uint32_t *instance*) [inline], [static]

This function enables the clock for PORT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.33 static void CLOCK_SYS_DisablePortClock (uint32_t *instance*) [inline], [static]

This function disables the clock for PORT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.34 static bool CLOCK_SYS_GetPortGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for PORT moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.35 static void CLOCK_SYS_EnableEwmClock (uint32_t *instance*) [inline], [static]

This function enables the clock for EWM moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.36 static void CLOCK_SYS_DisableEwmClock (uint32_t *instance*) [inline], [static]

This function disables the clock for EWM moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.37 static bool CLOCK_SYS_GetEwmGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for EWM moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.38 static void CLOCK_SYS_EnableFtfClock(uint32_t *instance*) [inline],
[static]**

This function enables the clock for FTF moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.39 static void CLOCK_SYS_DisableFtfClock (uint32_t *instance*) [inline], [static]

This function disables the clock for FTF moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.40 static bool CLOCK_SYS_GetFtfGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for FTF moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.41 static void CLOCK_SYS_EnableCrcClock (uint32_t *instance*) [inline], [static]

This function enables the clock for CRC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.42 static void CLOCK_SYS_DisableCrcClock (uint32_t *instance*) [inline], [static]

This function disables the clock for CRC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.43 static bool CLOCK_SYS_GetCrcGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for CRC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.44 static void CLOCK_SYS_EnableAdcClock (uint32_t *instance*) [inline], [static]

This function enables the clock for ADC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.45 static void CLOCK_SYS_DisableAdcClock (uint32_t *instance*) [inline], [static]

This function disables the clock for ADC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.46 static bool CLOCK_SYS_GetAdcGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for ADC moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.47 static void CLOCK_SYS_EnableCmpClock (uint32_t *instance*) [inline], [static]

This function enables the clock for CMP moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.48 static void CLOCK_SYS_DisableCmpClock (uint32_t *instance*) [inline], [static]

This function disables the clock for CMP moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.49 static bool CLOCK_SYS_GetCmpGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for CMP moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.50 static void CLOCK_SYS_EnableDacClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for DAC moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.51 static void CLOCK_SYS_DisableDacClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for DAC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.52 static bool CLOCK_SYS_GetDacGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for DAC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.53 static void CLOCK_SYS_EnableVrefClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for VREF moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.54 static void CLOCK_SYS_DisableVrefClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for VREF moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.55 static bool CLOCK_SYS_GetVrefGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for VREF moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.56 static void CLOCK_SYS_EnableSaiClock (uint32_t *instance*) [inline], [static]

This function enables the clock for SAI moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.57 static void CLOCK_SYS_DisableSaiClock (uint32_t *instance*) [inline], [static]

This function disables the clock for SAI moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.58 static bool CLOCK_SYS_GetSaiGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for SAI moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.59 static void CLOCK_SYS_EnablePdbClock (uint32_t *instance*) [inline], [static]

This function enables the clock for PDB moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.60 static void CLOCK_SYS_DisablePdbClock (uint32_t *instance*) [inline], [static]

This function disables the clock for PDB moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.61 static bool CLOCK_SYS_GetPdbGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for PDB moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.62 static void CLOCK_SYS_EnableFtmClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for FTM moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.63 static void CLOCK_SYS_DisableFtmClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for FTM moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.64 static bool CLOCK_SYS_GetFtmGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for FTM moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.65 static void CLOCK_SYS_EnablePitClock (uint32_t *instance*) [inline],
[static]**

This function enables the clock for PIT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.66 static void CLOCK_SYS_DisablePitClock (uint32_t *instance*) [inline],
[static]**

This function disables the clock for PIT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.67 static bool CLOCK_SYS_GetPitGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for PIT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.68 static void CLOCK_SYS_EnableLptimerClock (uint32_t *instance*) [inline], [static]

This function enables the clock for LPTIMER moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.69 static void CLOCK_SYS_DisableLptimerClock (uint32_t *instance*) [inline], [static]

This function disables the clock for LPTIMER moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.70 static bool CLOCK_SYS_GetLptimerGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for LPTIMER moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.71 static void CLOCK_SYS_EnableRtcClock (uint32_t *instance*) [inline], [static]

This function enables the clock for RTC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.72 static void CLOCK_SYS_DisableRtcClock (uint32_t *instance*) [inline], [static]

This function disables the clock for RTC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.73 static bool CLOCK_SYS_GetRtcGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for RTC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.74 static void CLOCK_SYS_EnableUsbClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for USBFS moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.75 static void CLOCK_SYS_DisableUsbClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for USBFS moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.76 static bool CLOCK_SYS_GetUsbGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for USB moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.77 static void CLOCK_SYS_EnableSpiClock (uint32_t *instance*) [inline],
[static]**

This function enables the clock for SPI moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.78 static void CLOCK_SYS_DisableSpiClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for SPI moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.79 static bool CLOCK_SYS_GetSpiGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for SPI moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.80 static void CLOCK_SYS_EnableI2cClock (uint32_t *instance*) [inline], [static]

This function enables the clock for I2C moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.81 static void CLOCK_SYS_DisableI2cClock (uint32_t *instance*) [inline], [static]

This function disables the clock for I2C moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.82 static bool CLOCK_SYS_GetI2cGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for I2C moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.83 static void CLOCK_SYS_EnableUartClock (uint32_t *instance*) [inline], [static]

This function enables the clock for UART moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.84 static void CLOCK_SYS_DisableUartClock (uint32_t *instance*) [inline], [static]

This function disables the clock for UART moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.85 static bool CLOCK_SYS_GetUartGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for UART moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.86 static void CLOCK_SYS_EnableLpuartClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for LPUART moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.87 static void CLOCK_SYS_DisableLpuartClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for LPUART moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.88 static bool CLOCK_SYS_GetLpuartGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for LPUART moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.89 uint32_t CLOCK_SYS_GetRngaFreq (uint32_t *instance*)

This function gets the clock frequency for RNGA module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for RNGA module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.90 static void CLOCK_SYS_EnableRngaClock (uint32_t *instance*) [inline], [static]

This function enables the clock for RNGA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.91 static void CLOCK_SYS_DisableRngaClock (uint32_t *instance*) [inline], [static]

This function disables the clock for RNGA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.92 static bool CLOCK_SYS_GetRngaGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for RNGA moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

Function Documentation

35.2.93 `uint32_t CLOCK_SYS_GetFlexbusFreq (uint32_t instance)`

This function gets the clock frequency for FLEXBUS moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for FLEXBUS module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.94 static void CLOCK_SYS_EnableFlexbusClock (uint32_t *instance*) [inline], [static]

This function enables the clock for FLEXBUS moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.95 static void CLOCK_SYS_DisableFlexbusClock (uint32_t *instance*) [inline], [static]

This function disables the clock for FLEXBUS moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.96 static bool CLOCK_SYS_GetFlexbusGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for FLEXBUS moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.97 uint32_t CLOCK_SYS_GetMpuFreq (uint32_t *instance*)

This function gets the clock frequency for MPU moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for MPU module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.98 uint32_t CLOCK_SYS_GetCmtFreq (uint32_t *instance*)

This function gets the clock frequency for CMT module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for CMT module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.99 **uint32_t CLOCK_SYS_GetEnetRmiiFreq (uint32_t *instance*)**

This function gets the clock frequency for ENET module RMII clock..

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for ENET module RMII clock.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.100 **uint32_t CLOCK_SYS_GetEnetTimeStampFreq (uint32_t *instance*)**

This function gets the clock frequency for ENET module TIME clock..

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for ENET module TIME clock.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.101 uint32_t CLOCK_SYS_GetUsbdcdfreq (uint32_t *instance*)

This function gets the clock frequency for USB DCD module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.102 uint32_t CLOCK_SYS_GetSdhcfreq (uint32_t *instance*)

This function gets the clock frequency for SDHC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

This function gets the clock frequency for SDHC module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

**35.2.103 static void CLOCK_SYS_EnableMpuClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for MPU moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.104 static void CLOCK_SYS_DisableMpuClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for MPU moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.105 static bool CLOCK_SYS_GetMpuGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for MPU moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.106 static void CLOCK_SYS_EnableCmtClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for CMT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.107 static void CLOCK_SYS_DisableCmtClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for CMT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.108 static bool CLOCK_SYS_GetCmtGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for CMT moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.109 static void CLOCK_SYS_EnableUsbdcClock (uint32_t *instance*) [inline], [static]

This function enables the clock for USBDCD moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.110 static void CLOCK_SYS_DisableUsbdcClock (uint32_t *instance*) [inline], [static]

This function disables the clock for USBDCD moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.111 static bool CLOCK_SYS_GetUsbdcGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for USBDCD moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.112 static void CLOCK_SYS_EnableFlexcanClock (uint32_t *instance*) [inline], [static]

This function enables the clock for FLEXCAN moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.113 static void CLOCK_SYS_DisableFlexcanClock (uint32_t *instance*) [inline], [static]

This function disables the clock for FLEXCAN moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.114 static bool CLOCK_SYS_GetFlexcanGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for FLEXCAN moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.115 static void CLOCK_SYS_EnableSdhcClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for SDHC moudle.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.116 static void CLOCK_SYS_DisableSdhcClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for SDHC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.117 static bool CLOCK_SYS_GetSdhcGateCmd (uint32_t *instance*)
[inline], [static]**

This function will get the clock gate state for SDHC moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

**35.2.118 static void CLOCK_SYS_EnableEnetClock (uint32_t *instance*)
[inline], [static]**

This function enables the clock for ENET moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

**35.2.119 static void CLOCK_SYS_DisableEnetClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for ENET moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.120 static bool CLOCK_SYS_GetEnetGateCmd (uint32_t *instance*) [inline], [static]

This function will get the clock gate state for ENET moudle.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.121 uint32_t CLOCK_SYS_GetTpmFreq (uint32_t *instance*)

This function gets the clock frequency for TPM module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

freq clock frequency for this module

35.2.122 static void CLOCK_SYS_EnableTpmClock (uint32_t *instance*) [inline], [static]

This function enables the clock for TPM module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Function Documentation

**35.2.123 static void CLOCK_SYS_DisableTpmClock (uint32_t *instance*)
[inline], [static]**

This function disables the clock for TPM module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.124 static bool CLOCK_SYS_GetTpmGateCmd (uint32_t *instance*) [inline], [static]

This function gets the clock gate state for TPM module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

35.2.125 static void CLOCK_SYS_EnableTsiClock (uint32_t *instance*) [inline], [static]

This function enables the clock for TSI module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.126 static void CLOCK_SYS_DisableTsiClock (uint32_t *instance*) [inline], [static]

This function disables the clock for TSI module.

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

35.2.127 static bool CLOCK_SYS_GetTsiGateCmd (uint32_t *instance*) [inline], [static]

This function gets the clock gate state for TSI module.

Function Documentation

Parameters

<i>instance</i>	module device instance
-----------------	------------------------

Returns

state true - ungated(Enabled), false - gated (Disabled)

Chapter 36

Hwtimer_driver

The Kinetis SDK provides the HwTimer driver for various timer modules.

36.0.128 HwTimer Driver

Overview

The HwTimer driver provides common API for various timer modules. The driver consists of two layers:

- The hardware-specific lower layer contains implementation specific for a particular timer module. An application should not use this layer.
- The generic upper layer provides abstraction to call the appropriate lower layer functions while passing the appropriate context structure to them. This chapter describes the generic upper layer only.

Chapter 37

Interrupt Manager (Interrupt)

The Kinetis SDK Interrupt Manager provides a set of API/services to configure the Interrupt Controller (NVIC).

interrupt_manager APIs

- void [INT_SYS_InstallHandler](#) (IRQn_Type irqNumber, void(*handler)(void))
Installs an interrupt handler routine for a given IRQ number.
- static void [INT_SYS_EnableIRQ](#) (IRQn_Type irqNumber)
Enables an interrupt for a given IRQ number.
- static void [INT_SYS_DisableIRQ](#) (IRQn_Type irqNumber)
Disables an interrupt for a given IRQ number.
- void [INT_SYS_EnableIRQGlobal](#) (void)
Enables system interrupt.
- void [INT_SYS_DisableIRQGlobal](#) (void)
Disable system interrupt.

37.0.129 Interrupt Manager

Overview

The Interrupt Manager provides a set of APIs so that the application can enable or disable an interrupt for a specific device and also set/get interrupt status, priority and other features. Also it provides a way to update the vector table for a specific device interrupt handler.

Interrupt Names

Each chip has its own set of supported interrupt names defined in the chip-specific header file. For example, for K70, the header file is MK70F12.h or MK70F15.h as an IRQn_Type.

Example to set/update the vector table for the I2C0_IRQn interrupt handler:

```
#include "interrupt/fsl_interrupt_manager.h"

interrupt_register_handler(I2C0_IRQn, irq_handler_I2C0_IRQn);
```

Example to enable/disable an interrupt for the I2C0_IRQn

```
#include "interrupt/fsl_interrupt_manager.h"

interrupt_enable(I2C0_IRQn);

interrupt_disable(I2C0_IRQn);
```

Function Documentation

37.1 Function Documentation

37.1.1 void INT_SYS_InstallHandler (IRQn_Type *irqNumber*, void(*)(void) *handler*)

This function lets the application register/replace the interrupt handler for a specified IRQ number. The IRQ number is different than the vector number. IRQ 0 starts from the vector 16 address. See a chip-specific reference manual for details and the startup_MKxxxx.s file for each chip family to find out the default interrupt handler for each device. This function converts the IRQ number to the vector number by adding 16 to it.

Parameters

<i>irqNumber</i>	IRQ number
<i>handler</i>	Interrupt handler routine address pointer

37.1.2 static void INT_SYS_EnableIRQ (IRQn_Type *irqNumber*) [inline], [static]

This function enables the individual interrupt for a specified IRQ number. It calls the system NVIC API to access the interrupt control register. The input IRQ number does not include the core interrupt, only the peripheral interrupt, from 0 to a maximum supported IRQ.

Parameters

<i>irqNumber</i>	IRQ number
------------------	------------

37.1.3 static void INT_SYS_DisableIRQ (IRQn_Type *irqNumber*) [inline], [static]

This function disables the individual interrupt for a specified IRQ number. It calls the system NVIC API to access the interrupt control register.

Parameters

<i>irqNumber</i>	IRQ number
------------------	------------

37.1.4 void INT_SYS_EnableIRQGlobal (void)

This function enables the global interrupt by calling the core API.

37.1.5 void INT_SYS_DisableIRQGlobal(void)

This function disables the global interrupt by calling the core API.

Function Documentation

Chapter 38

Power Manager (Power)

The Kinetis SDK Power Manager provides a set of API/services to configure the power-related IPs, such as SMC, PMC, RCM, etc.

Data Structures

- struct [power_manager_user_config_t](#)
Power mode user configuration structure. [More...](#)
- struct [power_manager_static_callback_user_config_t](#)
Statically allocated callback. [More...](#)
- struct [power_manager_dynamic_callback_user_config_t](#)
Dynamically allocated callback. [More...](#)
- struct [power_manager_state_t](#)
Power manager internal state structure. [More...](#)

Macros

- #define [POWER_SYS_CALLBACK_BEFORE](#) 1
Internal constant.
- #define [POWER_SYS_CALLBACK_AFTER](#) 2
Internal constant.

Typedefs

- typedef uint8_t [power_manager_callback_priority_t](#)
Callback invocation priority.
- typedef uint32_t [power_manager_callback_handle_t](#)
Callback handle.
- typedef void [power_manager_callback_data_t](#)
Callback-specific data.
- typedef [power_manager_error_code_t](#)(* [power_manager_callback_t](#))[\(power_manager_callback_type_t](#)
type, [power_manager_user_config_t](#) *configPtr, [power_manager_callback_data_t](#) *dataPtr)
Callback prototype.

Enumerations

- enum [power_manager_modes_t](#)
Power modes enumeration.
- enum [power_manager_error_code_t](#)
Power manager success code and error codes.
- enum [power_manager_policy_t](#)
Power manager policies.
- enum [power_manager_callback_type_t](#)
Callback invocation types.

Functions

- `power_manager_error_code_t POWER_SYS_Init (power_manager_user_config_t **(powerConfigsPtr)[], uint8_t configsNumber, power_manager_static_callback_user_config_t **(callbacksPtr)[], uint8_t callbacksNumber)`
Power manager initialization for operation.
- `power_manager_error_code_t POWER_SYS_Deinit (void)`
This function deinitializes the Power manager.
- `power_manager_error_code_t POWER_SYS_SetMode (uint8_t powerModeIndex)`
This function configures the power mode.
- `power_manager_error_code_t POWER_SYS_GetMode (uint8_t *powerModeIndexPtr)`
This function returns power mode set as the last one.
- `power_manager_error_code_t POWER_SYS_GetModeConfig (power_manager_user_config_t **powerModePtr)`
This function returns user configuration structure of power mode set as the last one.
- `power_manager_modes_t POWER_SYS_GetRunningMode (void)`
This function returns currently running power mode.
- `power_manager_callback_handle_t POWER_SYS_GetErroneousDriver (void)`
This function returns the last failed notification callback.
- `power_manager_error_code_t POWER_SYS_RegisterCallbackFunction (power_manager_dynamic_callback_user_config_t *callbackPtr, power_manager_callback_handle_t *callbackHandlePtr)`
This function registers notification callback.
- `power_manager_error_code_t POWER_SYS_UnregisterCallbackFunction (power_manager_callback_handle_t callbackHandle)`
This function unregisters specified notification callback.
- `bool POWER_SYS_GetVeryLowPowerModeStatus (void)`
This function returns whether very low power mode is running.
- `bool POWER_SYS_GetLowLeakageWakeupResetStatus (void)`
This function returns whether reset was caused by low leakage wake up.

38.0.6 Power Manager

Overview

Low Power Manager provides API to handle the device power modes. It also supports run-time switching between multiple power modes. Each power mode is described by configuration structures with multiple power-related options. Low Power Manager provides a notification mechanism for registered callbacks and API for static and dynamic callback registration.

The Power Manager driver is developed on top of the SMC HAL, PMC HAL and RCM HAL.

This is an example to initialize the Power Manager:

~~~~~{.c}

```
#include "fsl_power_manager.h"
```



This is an example to switch into a desired power mode:

```
~~~~~{.c}

#include "fsl_power_manager.h"

/* Index into array containing configuration of very low power run mode - vlpr */
#define MODE_VLPR 0U

/* Initialization */
power_manager_user_config_t vlprConfig;
...
...
power_manager_user_config_t *powerConfigs[1] = {&vlprConfig};

/* Switch to required power mode */
power_manager_error_code_t ret = POWER_SYS_SetMode(MODE_VLPR);

if (ret != kPowerManagerSuccess)
{
 printf("POWER_SYS_SetMode(powerMode5) returns : %u\n\r", ret);
}
```

~~~~~{.c}

### 38.1 Data Structure Documentation

#### 38.1.1 struct power\_manager\_user\_config\_t

This structure defines Kinetis power mode with additional power options and specifies transition to and out of this mode. Application may define multiple power modes and switch between them. List of defined power modes is passed to the Power manager during initialization as an array of references to structures of this type (see [POWER\\_SYS\\_Init\(\)](#)). Power modes can be switched by calling [POWER\\_SYS\\_SetMode\(\)](#) which accepts index to the list of power modes passed during manager initialization. Currently used power mode can be retrieved by calling [POWER\\_SYS\\_GetMode\(\)](#), which returns index of the current power mode, or by [POWER\\_SYS\\_GetModeConfig\(\)](#), which returns reference to the structure of current mode. List of power mode configuration structure members depends on power options available for specific chip. Complete list contains: mode - Kinetis power mode. List of available modes is chip-specific. See power\_manager\_modes\_t list of modes. This item is common for all Kinetis chips. policy - Power mode change policy. Specifies whether the callbacks notified before the power mode change can prohibit the switch or callback notification results are ignored forcing the switch. This item is common for all Kinetis chips. sleepOnExitOption - Controls whether the sleep-on-exit option value is used (when set to true) or ignored (when set to false). See sleepOnExitValue. This item is common for all Kinetis chips. sleepOnExitValue - When set to true, ARM core returns to sleep (Kinetis wait modes) or deep sleep state (Kinetis stop modes) after interrupt service finishes. When set to false, core stays woken-up. This item is common for all Kinetis chips. lowPowerWakeUpOnInterruptOption - Controls whether the wake-up-on-interrupt option value is used (when set to true) or ignored (when set to false). See lowPowerWakeUpOnInterruptOption. This item is chip-specific. sleepOnExitValue - When set to true, system exits to Run mode when

any interrupt occurs while in Very low power run, Very low power wait or Very low power stop mode. This item is chip-specific. `powerOnResetDetectionOption` - Controls whether the power on reset detection option value is used (when set to true) or ignored (when set to false). See `powerOnResetDetectionOption`. This item is chip-specific. `powerOnResetDetectionValue` - When set to true, power on reset detection circuit is enabled in Very low leakage stop 0 mode. When set to false, circuit is disabled. This item is chip-specific. `RAM2PartitionOption` - Controls whether the RAM2 partition power option value is used (when set to true) or ignored (when set to false). See `RAM2PartitionValue`. This item is chip-specific. `RAM2PartitionValue` - When set to true, RAM2 partition content is retained through Very low leakage stop 2 mode. When set to false, RAM2 partition power is disabled and memory content lost. This item is chip-specific. `lowPowerOscillatorOption` - Controls whether the Low power oscillator power option value is used (when set to true) or ignored (when set to false). See `lowPowerOscillatorValue`. This item is chip-specific. `lowPowerOscillatorValue` - When set to true, the 1 kHz Low power oscillator is enabled in any Low leakage or Very low leakage stop mode. When set to false, oscillator is disabled in these modes. This item is chip-specific.

### **38.1.2 struct power\_manager\_static\_callback\_user\_config\_t**

This structure holds configuration of callbacks passed to the Power manager during its initialization. Callbacks of this type are expected to be statically allocated. This structure contains following application-defined data: `callback` - pointer to the callback function `callbackType` - specifies when the callback is called `callbackData` - pointer to the data passed to the callback

### **38.1.3 struct power\_manager\_dynamic\_callback\_user\_config\_t**

This structure holds configuration of callbacks registered to the Power manager during its run-time through [`POWER\_SYS\_RegisterCallbackFunction\(\)`](#). Dynamically registered callbacks are managed as a double linked list where link order is handled by the Power manager. This structure contains both application-defined data and data handled by the Power manager: `callback` - pointer to the callback function, defined by the application `callbackType` - specifies when the callback is called, defined by the application `callbackData` - pointer to the data passed to the callback, defined by the application `callbackPriority` - specifies order in which are callbacks called, defined by the application `handle` - callback ID, defined by the Power manager `prev` - previous list node reference, defined by the Power manager `next` - next list node reference, defined by the Power manager

### **38.1.4 struct power\_manager\_state\_t**

Power manager internal structure. Contains data necessary for Power manager proper function. Stores references to registered power mode configurations, statically and dynamically registered callbacks, information about their numbers and other internal data. This structure is statically allocated and initialized after [`POWER\_SYS\_Init\(\)`](#) call. It contains: `configs` - Reference to array of power mode configuration structures. Filled once during [`POWER\_SYS\_Init\(\)`](#) call. `staticCallbacks` - Reference to array of statically registered callbacks. Filled once during [`POWER\_SYS\_Init\(\)`](#) call. `dynamicCallbacks` - Reference to linked

## Typedef Documentation

list of dynamically registered callbacks. Accessed during [POWER\\_SYS\\_RegisterCallbackFunction\(\)](#) and [POWER\\_SYS\\_UnregisterCallbackFunction\(\)](#) calls. callbacksHandleCounter - Internal generator of callback handles. lastErroneousHandle - Stores handle of last callback which failed during notification call while running [POWER\\_SYS\\_SetMode\(\)](#). configsNumber - Number of power mode configuration passed during Power manager initialization. Size of array reference by configs. staticCallbacksNumber - Number of statically registered callbacks passed during Power manager initialization. Size of array reference by staticCallbacks. dynamicCallbacksNumber - Number of dynamically registered callbacks. Size of linked list referenced by dynamicCallbacks. currentConfig - Power mode set as the last one. Index to the array referenced by configs.

## 38.2 Macro Definition Documentation

### 38.2.1 #define POWER\_SYS\_CALLBACK\_BEFORE 1

Used to identify callbacks invoked before the power mode change. See `power_manager_callback_type_t`.

### 38.2.2 #define POWER\_SYS\_CALLBACK\_AFTER 2

Used to identify callbacks invoked after the power mode change. See `power_manager_callback_type_t`.

## 38.3 Typedef Documentation

### 38.3.1 typedef uint8\_t power\_manager\_callback\_priority\_t

Used in the dynamically registered callback configuration structure ([power\\_manager\\_dynamic\\_callback\\_user\\_config\\_t](#)). Defines invocation order of the registered callbacks (callbacks registered by the [POWER\\_SYS\\_RegisterCallbackFunction\(\)](#)). Lower number has higher priority. In cases of multiple callbacks registered with the same priority, the registration order is decisive (first registered is first serviced). Supported priorities are in range of 0-255, however, priorities in range 0-7 are reserved for system purposes. Statically registered callbacks ([power\\_manager\\_static\\_callback\\_user\\_config\\_t](#)) passed during Power manager initialization (see [POWER\\_SYS\\_Init\(\)](#)) have fixed priority and are always serviced after the dynamically registered callbacks (as if they had priority 256).

### 38.3.2 typedef uint32\_t power\_manager\_callback\_handle\_t

Defines unique identification of registered callback. Used for last failed callback identification (see [POWER\\_SYS\\_GetErroneousDriver\(\)](#)) and callback registration removal (see [POWER\\_SYS\\_UnregisterCallbackFunction\(\)](#)). For statically registered callbacks (see [POWER\\_SYS\\_Init\(\)](#)) the handle is equal to callback position in the array of registered callbacks counted from 1 (e.g. if two callbacks are registered the callback with index 0 has handle 1 and callback with index 1 has handle 2). For dynamically registered

callbacks the handle is returned by [POWER\\_SYS\\_RegisterCallbackFunction\(\)](#) and is part of [power\\_manager\\_dynamic\\_callback\\_user\\_config\\_t](#) structure. Handlers generation is reset after [POWER\\_SYS\\_Init\(\)](#) call. Valid handle has non-zero, 32-bit value.

### 38.3.3 **typedef void power\_manager\_callback\_data\_t**

Reference to data of this type is passed during callback registration. The reference is part of the [power\\_manager\\_static\\_callback\\_user\\_config\\_t](#) or [power\\_manager\\_dynamic\\_callback\\_user\\_config\\_t](#) structure and is passed to the callback during power mode change notifications (see [POWER\\_SYS\\_SetMode\(\)](#) and [power\\_manager\\_callback\\_t](#)).

### 38.3.4 **typedef power\_manager\_error\_code\_t(\* power\_manager\_callback\_t)(power\_manager\_callback\_type\_t type, power\_manager\_user\_config\_t \*configPtr, power\_manager\_callback\_data\_t \*dataPtr)**

Declaration of callback. It is common for statically and dynamically registered callbacks. Reference to function of this type is part of [power\\_manager\\_static\\_callback\\_user\\_config\\_t](#) and [power\\_manager\\_dynamic\\_callback\\_user\\_config\\_t](#) callback configuration structure. Depending on callback type, function of this prototype is called during power mode change (see [POWER\\_SYS\\_SetMode\(\)](#)) before the mode change, after it or in both cases to notify about the change progress (see [power\\_manager\\_callback\\_type\\_t](#)). When called, type of the notification is passed as parameter along with reference to entered power mode configuration structure (see [power\\_manager\\_user\\_config\\_t](#)) and any data passed during the callback registration (see [power\\_manager\\_callback\\_data\\_t](#)). When notified before the mode change, depending on the power mode change policy (see [power\\_manager\\_policy\\_t](#)) the callback may deny the mode change by returning any error code different from kPowerManagerSuccess (see [POWER\\_SYS\\_SetMode\(\)](#)).

Parameters

|                  |                                                                                                                                                                                                                                   |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>type</i>      | Notification type. Denotes whether the callback is invoked before the power mode change (then the type has value kPowerManagerCallbackBefore) or after successful or unsuccessful mode change (value kPowerManagerCallbackAfter). |
| <i>configPtr</i> | Entered power mode. Refers to the power mode configuration structure which contains settings of now entered (when type is kPowerManagerCallbackBefore) or exited mode (when type is kPowerManagerCallbackAfter).                  |

## Enumeration Type Documentation

|                |                                                                                                                                                            |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>dataPtr</i> | Callback data. Refers to the data passed during callback registration. Intended to pass any driver or application data such as internal state information. |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|

Returns

An error code or kPowerManagerSuccess.

## 38.4 Enumeration Type Documentation

### 38.4.1 enum power\_manager\_modes\_t

Defines power mode. Used in the power mode configuration structure ([power\\_manager\\_user\\_config\\_t](#)). From ARM core perspective, Power modes can be generally divided to run modes (High speed run, Run and Very low power run), sleep (Wait and Very low power wait) and deep sleep modes (all Stop modes). List of power modes supported by specific chip along with requirements for entering and exiting of these modes can be found in chip documentation. List of all supported power modes: kPowerManagerHsrun - High speed run mode. Chip-specific. kPowerManagerRun - Run mode. All Kinetis chips. kPowerManagerVlpr - Very low power run mode. All Kinetis chips. kPowerManagerWait - Wait mode. All Kinetis chips. kPowerManagerVlpw - Very low power wait mode. All Kinetis chips. kPowerManagerStop - Stop mode. All Kinetis chips. kPowerManagerVlps - Very low power stop mode. All Kinetis chips. kPowerManagerPstop1 - Partial stop 1 mode. Chip-specific. kPowerManagerPstop2 - Partial stop 2 mode. Chip-specific. kPowerManagerLls - Low leakage stop mode. All Kinetis chips. kPowerManagerLls2 - Low leakage stop 2 mode. Chip-specific. kPowerManagerLls3 - Low leakage stop 3 mode. Chip-specific. kPowerManagerVlls0 - Very low leakage stop 0 mode. All Kinetis chips. kPowerManagerVlls1 - Very low leakage stop 1 mode. All Kinetis chips. kPowerManagerVlls2 - Very low leakage stop 2 mode. All Kinetis chips. kPowerManagerVlls3 - Very low leakage stop 3 mode. All Kinetis chips.

### 38.4.2 enum power\_manager\_error\_code\_t

Used as return value of Power manager functions.

### 38.4.3 enum power\_manager\_policy\_t

Define whether the power mode change is forced or not. Used in the power mode configuration structure ([power\\_manager\\_user\\_config\\_t](#)) to specify whether the mode switch initiated by the [POWER\\_SYS\\_SetMode\(\)](#) depends on the callback notification results. For kPowerManagerPolicyForcible the power mode is changed regardless of the results, while kPowerManagerPolicyAgreement policy is used the [POWER\\_SYS\\_SetMode\(\)](#) is exited when any of the callbacks returns error code. See also [POWER\\_SYS\\_SetMode\(\)](#) description.

### 38.4.4 enum power\_manager\_callback\_type\_t

Used in the callback configuration structures ([power\\_manager\\_static\\_callback\\_user\\_config\\_t](#) and [power\\_manager\\_dynamic\\_callback\\_user\\_config\\_t](#)) to specify when the registered callback will be called during power mode change initiated by [POWER\\_SYS\\_SetMode\(\)](#). Also used when callback is invoked to specify notification type. Callback can be invoked in following situations:

- before the power mode change (Callback return value can affect [POWER\\_SYS\\_SetMode\(\)](#) execution. Refer to the [POWER\\_SYS\\_SetMode\(\)](#) and [power\\_manager\\_policy\\_t](#) documentation).
- after entering one of the run modes or after exiting from one of the (deep) sleep power modes back to the run mode.
- after unsuccessful attempt to switch power mode Values used during callback registration:
- kPowerManagerCallbackBefore
- kPowerManagerCallbackAfter
- kPowerManagerCallbackBeforeAfter Values used during callback invocation:
- kPowerManagerCallbackBefore
- kPowerManagerCallbackAfter
- kPowerManagerCallbackFailed

## 38.5 Function Documentation

### 38.5.1 power\_manager\_error\_code\_t POWER\_SYS\_Init ( [power\\_manager\\_user\\_config\\_t](#) \*[\(\\*\) powerConfigsPtr\[\]](#), [uint8\\_t](#) [configsNumber](#), [power\\_manager\\_static\\_callback\\_user\\_config\\_t](#) \*[\(\\*\) callbacksPtr\[\]](#), [uint8\\_t](#) [callbacksNumber](#) )

This function initializes the Power manager and its run-time state structure. Reference to an array of Power mode configuration structures has to be passed as a parameter along with a parameter specifying its size. At least one power mode configuration is required. Optionally, reference to the array of predefined callbacks can be passed with its size parameter. This can be used to pass callbacks directly without need of dynamic, run-time registration (see [POWER\\_SYS\\_RegisterCallbackFunction\(\)](#) and [POWER\\_SYS\\_UnregisterCallbackFunction\(\)](#)). For details about static callbacks, refer to the [power\\_manager\\_static\\_callback\\_user\\_config\\_t](#). As Power manager stores only references to array of these structures, they have to exist while Power manager is used. It is expected that prior to the [POWER\\_SYS\\_Init\(\)](#) call the write-once protection register was configured appropriately allowing for entry to all required low power modes. The following is an example of how to set up two power modes and three callbacks, and initialize the Power manager with structures containing their settings. The example shows two possible ways the configuration structures can be stored (ROM or RAM), although it is expected that they will be placed in the read-only memory to save the RAM space. (Note: In the example it is assumed that the programmed chip doesn't support any optional power options described in the [power\\_manager\\_user\\_config\\_t](#) ) :

```
const power_manager_user_config_t waitConfig = {
 kPowerManagerVlpw,
 kPowerManagerPolicyForcible,
 true,
 true,
};
```

## Function Documentation

```
const power_manager_static_callback_user_config_t
callbackCfg0 = {
 callback0,
 kPowerManagerCallbackBefore,
 &callback_data0
};

const power_manager_static_callback_user_config_t
callbackCfg1 = {
 callback1,
 kPowerManagerCallbackAfter,
 &callback_data1
};

const power_manager_static_callback_user_config_t
callbackCfg2 = {
 callback2,
 kPowerManagerCallbackBeforeAfter,
 &callback_data2
};

const power_manager_static_callback_user_config_t * const
callbacks[] = {&callbackCfg0, &callbackCfg1, &callbackCfg2};

void main(void)
{
 power_manager_user_config_t idleConfig;
 power_manager_user_config_t *powerConfigs[POWER_SYS_CONFIGURATIONS] = {&
idleConfig, &idleConfig};

 idleConfig.mode = kPowerManagerVlps;
 idleConfig.policy = kPowerManagerPolicyForcible;
 idleConfig.sleepOnExitOption = true;
 idleConfig.sleepOnExitValue = false;

 POWER_SYS_Init(&powerConfigs, 2U, &callbacks, 3U);

 POWER_SYS_SetMode(0U);

}
*
```

### Parameters

|                              |                                                                                                                                                                                                                              |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>powerConfigs-<br/>Ptr</i> | A pointer to an array with references to all power configurations which will be handled by Power manager.                                                                                                                    |
| <i>configsNumber</i>         | Number of power configurations. Size of powerConfigsPtr array.                                                                                                                                                               |
| <i>callbacksPtr</i>          | A pointer to an array with references to callback configurations. These callbacks are statically registered in the Power manager. If there are no callbacks to register during Power manager initialization, use NULL value. |
| <i>callbacks-<br/>Number</i> | Number of statically registered callbacks. Size of callbacksPtr array.                                                                                                                                                       |

### Returns

An error code or kPowerManagerSuccess.

### 38.5.2 power\_manager\_error\_code\_t POWER\_SYS\_Deinit( void )

Returns

An error code or kPowerManagerSuccess.

### 38.5.3 power\_manager\_error\_code\_t POWER\_SYS\_SetMode( uint8\_t powerModelIndex )

This function switches to one of the defined power modes. Requested mode number is passed as an input parameter. This function notifies all registered callback functions before the mode change (using kPowerManagerCallbackBefore set as callback type parameter), sets specific power options defined in the power mode configuration and enters the specified mode. In case of run modes (for example, Run, Very low power run, or High speed run), this function also invokes all registered callbacks after the mode change (using kPowerManagerCallbackAfter). In case of sleep or deep sleep modes, if the requested mode is not exited through a reset, these notifications are sent after the core wakes up. Callbacks are invoked in the following order: first, all dynamically registered callbacks are invoked ordered by the priority set during their registrations (see POWER\_SYS\_RegisterCallbackFunction). Then all statically registered callbacks are notified ordered by index in the static callbacks array (see callbacksPtr parameter of [POWER\\_SYS\\_Init\(\)](#)). The same order is used for before and after switch notifications. The notifications before the power mode switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the power mode change, further execution of this function depends on mode change policy defined in the user power mode configuration (see policy in the [power\\_manager\\_user\\_config\\_t](#)): the mode change is either forced (kPowerManagerPolicyForcible) or exited (kPowerManagerPolicyAgreement). When mode change is forced, the result of the before switch notifications are ignored. If agreement is required, if any callback returns an error code then further notifications before switch notifications are cancelled and all already notified callbacks are re-invoked with kPowerManagerCallbackAfter set as callback type parameter. The handler of the callback which returned error code during pre-switch notifications is stored (any error codes during callbacks re-invocation are ignored) and [POWER\\_SYS\\_GetErroneousDriver\(\)](#) can be used to get it. Regardless of the policies, if any callback returned an error code, an error code denoting in which phase the error occurred is returned when [POWER\\_SYS\\_SetMode\(\)](#) exits. It is possible to enter any mode supported by the processor. Refer to the chip reference manual for list of available power modes. If it is necessary to switch into intermediate power mode prior to entering requested mode (for example, when switching from Run into Very low power wait through Very low power run mode), then the intermediate mode is entered without invoking the callback mechanism.

Parameters

|                       |                                                                                                                                                    |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>powerModeIndex</i> | Requested power mode represented as an index into array of user-defined power mode configurations passed to the <a href="#">POWER_SYS_Init()</a> . |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|

## Function Documentation

Returns

An error code or kPowerManagerSuccess.

### 38.5.4 power\_manager\_error\_code\_t POWER\_SYS\_GetMode ( uint8\_t \* *powerModelIndexPtr* )

This function returns index of power mode which was set using [POWER\\_SYS\\_SetMode\(\)](#) as the last one. If the power mode was entered even though some of the registered callback denied the mode change, or if any of the callbacks invoked after the entering/restoring run mode failed, then the return code of this function has kPowerManagerError value.

Parameters

|                        |                                                                                                                                                |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>powerModelIndex</i> | Power mode which has been set represented as an index into array of power mode configurations passed to the <a href="#">POWER_SYS_Init()</a> . |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|

Returns

An error code or kPowerManagerSuccess.

### 38.5.5 power\_manager\_error\_code\_t POWER\_SYS\_GetModeConfig ( power\_manager\_user\_config\_t \*\* *powerModePtr* )

This function returns reference to configuration structure which was set using [POWER\\_SYS\\_SetMode\(\)](#) as the last one. If the current power mode was entered even though some of the registered callback denied the mode change, or if any of the callbacks invoked after the entering/restoring run mode failed, then the return code of this function has kPowerManagerError value.

Parameters

|                           |                                                                              |
|---------------------------|------------------------------------------------------------------------------|
| <i>powerModelIndexPtr</i> | Pointer to power mode configuration structure of power mode set as last one. |
|---------------------------|------------------------------------------------------------------------------|

Returns

An error code or kPowerManagerSuccess.

### 38.5.6 power\_manager\_modes\_t POWER\_SYS\_GetRunningMode ( void )

This function reads hardware settings and returns currently running power mode. Generally, this function can return only kPowerManagerRun, kPowerManagerVlpr or kPowerManagerHsrun value.

Returns

Currently used run power mode.

### **38.5.7 power\_manager\_callback\_handle\_t POWER\_SYS\_GetErroneousDriver ( void )**

This function returns handle of the last callback that failed during the power mode switch while the last [POWER\\_SYS\\_SetMode\(\)](#) was called. If the last [POWER\\_SYS\\_SetMode\(\)](#) call ended successfully, zero value is returned. Returned callback ID represents either the handle assigned to the callback during its dynamic registration (the handle return parameter of the [POWER\\_SYS\\_RegisterCallbackFunction\(\)](#)) or, in cases of statically registered callbacks, index in the array of static callbacks counted from 1 (for example, static callback at index 0 of the array has handle value 1, index 1 is represented by handle value 2 etc.)

Returns

Zero or callback function handle.

### **38.5.8 power\_manager\_error\_code\_t POWER\_SYS\_RegisterCallbackFunction ( power\_manager\_dynamic\_callback\_user\_config\_t \* callbackPtr, power\_manager\_callback\_handle\_t \* callbackHandlePtr )**

This function is used to dynamically register driver or application callback function into the Power manager. Callbacks are used during power mode switch to notify and receive acknowledge from registered functions: all functions which were registered with kPowerManagerCallbackBefore or kPowerManagerCallbackBeforeAfter callback type (see [power\\_manager\\_dynamic\\_callback\\_user\\_config\\_t](#) and [power\\_manager\\_callback\\_type\\_t](#)) are called before the power mode is switched. It is expected that these callbacks will acknowledge readiness for the power mode change. If any callback signals that it is not ready for the change, then subsequent mode change execution depends on the policy of the requested power mode configuration structure (see [power\\_manager\\_policy\\_t](#)). When the power mode is switched (in cases of run modes) or exited without a reset (in cases of sleep modes) then callbacks registered with kPowerManagerCallbackAfter or kPowerManagerCallbackBeforeAfter are invoked. Registered callback is invoked during each [POWER\\_SYS\\_SetMode\(\)](#) call. An order of invocation of callbacks is specified by priority number (see [power\\_manager\\_callback\\_priority\\_t](#)). Callbacks with lower priority number are called first. In cases of multiple callbacks having the same priority, the first registered callback is serviced as first (FIFO). When called, reference to [power\\_manager\\_callback\\_data\\_t](#) registered with the callback are passed to the callback by the power manager, along with reference to power mode configuration reference to provide callback with all necessary information about power mode which being entered. Following successful registration, a handle identifying the callback is returned. This handle is used to identify last unsuccessfully invoked callback (see [POWER\\_SYS\\_GetErroneousDriver\(\)](#)) and for callback unregistration (see [POWER\\_SYS\\_UnregisterCallbackFunction](#)). See [power\\_manager\\_callback\\_t](#) for required callback prototype. Following is an example of usage:

## Function Documentation

```
struct {

 //...

} callbackData;

power_manager_error_code_t callback(
 power_manager_callback_type_t type,
 power_manager_user_config_t *config,
 power_manager_callback_data_t *dataPtr)
{
 switch (type) {
 case kPowerManagerCallbackBefore:
 if (config->mode <= kPowerManagerVlpr)
 {
 // Run modes
 //...
 }
 else
 {
 // Sleep/deep sleep modes
 //...
 }
 break;
 case kPowerManagerCallbackAfter:
 if (POWER_SYS_GetRunningMode() == kPowerManagerRun)
 {
 //...
 }
 else
 {
 //...
 }
 break;
 }
 return kPowerManagerSuccess;
}

void main (void) {
 power_manager_error_code_t registrationStatus;
 power_manager_callback_handle_t callbackHandle;
 power_manager_dynamic_callback_user_config_t
callbackDescriptor = {
 &callback,
 kPowerManagerCallbackBeforeAfter,
 &callbackData,
 255,
 0,
 NULL,
 NULL
};

//Power manager initialization
//...

registrationStatus = POWER_SYS_RegisterCallbackFunction(&
callbackDescriptor, &callbackHandle);
}
*
```

Parameters

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>callbackPtr</i>       | Structure with callback configuration. Used to specify callback address (power_manager_callback_t callback), type (power_manager_callback_type_t callbackType), data (power_manager_callback_data_t callbackData) and priority (power_manager_callback_priority_t callbackPriority). The rest of the structure content is used by the Power manager: the unique handle assigned to the callback and linked list references. Creation of this structure, including memory allocation, is managed by the application; only reference is stored in the Power manager therefore it has to be available during Power manager usage. If callback and its configuration structure has to be destroyed/deallocated it has to be unregistered from Power manager first. |
| <i>callbackHandlePtr</i> | Output parameter to return unique handle of registered callback. Only non-zero values are valid.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

Returns

An error code or kPowerManagerSuccess.

### 38.5.9 **power\_manager\_error\_code\_t POWER\_SYS\_UnregisterCallbackFunction ( power\_manager\_callback\_handle\_t *callbackHandle* )**

This function is used to unregister callback from the Power manager. After that, callback is no longer invoked. Callback to be removed is identified by handle received during its registration. If no registered callback matches the handle, kPowerManagerOutOfRangeError is returned. Only dynamically registered callbacks can be unregistered. This function can't be used to stop invocation of statically registered callbacks.

Parameters

|                       |                                            |
|-----------------------|--------------------------------------------|
| <i>callbackHandle</i> | Handle identifying callback to be removed. |
|-----------------------|--------------------------------------------|

Returns

An error code or kPowerManagerSuccess.

### 38.5.10 **bool POWER\_SYS\_GetVeryLowPowerModeStatus ( void )**

This function is used to detect whether very low power mode is running.

Returns

Returns true if processor runs in very low power mode, otherwise false.

## Function Documentation

### 38.5.11 **bool POWER\_SYS\_GetLowLeakageWakeupResetStatus ( void )**

This function is used to check that processor exited low leakage power mode through reset.

Returns

Returns true if processor was reset by low leakage wake up, otherwise false.

# Chapter 39

## Utilities for the Kinetis SDK

### Modules

- [Debug\\_console](#)

*This part describes the programming interface of the debug console driver.*

## **Debug\_console**

### **39.1 Debug\_console**

This chapter describes the programming interface of the debug console driver.

# Chapter 40

## OS Abstraction Layer (OSA)

The Kinetis SDK provides the timer services for all OSA layers.

### Modules

- [Bare Metal Abstraction Layer](#)

*The Kinetis SDK provides the Bare Metal Abstraction Layer for synchronization, mutual exclusion, message queue, etc.*

- [FreeRTOS Abstraction Layer](#)

*The Kinetis SDK provides the FreeRTOS Abstraction Layer for synchronization, mutual exclusion, message queue, etc.*

- [MQX Abstraction Layer](#)

*The Kinetis SDK provides the MQX Abstraction Layer for synchronization, mutual exclusion, message queue, etc.*

- [μC/OS-II Abstraction Layer](#)

*The Kinetis SDK provides the μC/OS-II Abstraction Layer for synchronization, mutual exclusion, message queue, etc.*

- [μC/OS-III Abstraction Layer](#)

*The Kinetis SDK provides the μC/OS-III Abstraction Layer for synchronization, mutual exclusion, message queue, etc.*

### Enumerations

- enum [osa\\_status\\_t](#) {  
    kStatus\_OSA\_Success = 0U,  
    kStatus\_OSA\_Error = 1U,  
    kStatus\_OSA\_Timeout = 2U,  
    kStatus\_OSA\_Idle = 3U }

*Defines the return status of OSA's functions.*

- enum [osa\\_event\\_clear\\_mode\\_t](#) {  
    kEventAutoClear = 0U,  
    kEventManualClear = 1U }

*The event flags are cleared automatically or manually.*

- enum [osa\\_critical\\_part\\_mode\\_t](#) {  
    kCriticalLockSched = 0U,  
    kCriticalDisableInt = 1U }

*Locks the task scheduler or disables interrupt in critical part.*

### Counting Semaphore

- [osa\\_status\\_t OSA\\_SemaCreate \(semaphore\\_t \\*pSem, uint8\\_t initialValue\)](#)  
*Creates a semaphore with a given value.*
- [osa\\_status\\_t OSA\\_SemaWait \(semaphore\\_t \\*pSem, uint32\\_t timeout\)](#)

- Pending a semaphore with timeout.
- **osa\_status\_t OSA\_SemaPost (semaphore\_t \*pSem)**  
Signals for someone waiting on the semaphore to wake up.
- **osa\_status\_t OSA\_SemaDestroy (semaphore\_t \*pSem)**  
Destroys a previously created semaphore.

## Mutex

- **osa\_status\_t OSA\_MutexCreate (mutex\_t \*pMutex)**  
Create an unlocked mutex.
- **osa\_status\_t OSA\_MutexLock (mutex\_t \*pMutex, uint32\_t timeout)**  
Waits for a mutex and locks it.
- **osa\_status\_t OSA\_MutexUnlock (mutex\_t \*pMutex)**  
Unlocks a previously locked mutex.
- **osa\_status\_t OSA\_MutexDestroy (mutex\_t \*pMutex)**  
Destroys a previously created mutex.

## Event signalling

- **osa\_status\_t OSA\_EventCreate (event\_t \*pEvent, osa\_event\_clear\_mode\_t clearMode)**  
Initializes an event object with all flags cleared.
- **osa\_status\_t OSA\_EventWait (event\_t \*pEvent, event\_flags\_t flagsToWait, bool waitAll, uint32\_t timeout, event\_flags\_t \*setFlags)**  
Waits for specified event flags to be set.
- **osa\_status\_t OSA\_EventSet (event\_t \*pEvent, event\_flags\_t flagsToSet)**  
Sets one or more event flags.
- **osa\_status\_t OSA\_EventClear (event\_t \*pEvent, event\_flags\_t flagsToClear)**  
Clears one or more flags.
- **osa\_status\_t OSA\_EventDestroy (event\_t \*pEvent)**  
Destroys a previously created event object.

## Task management

- **osa\_status\_t OSA\_TaskCreate (task\_t task, uint8\_t \*name, uint16\_t stackSize, task\_stack\_t \*stackMem, uint16\_t priority, task\_param\_t param, bool usesFloat, task\_handler\_t \*handler)**  
Creates a task.
- **osa\_status\_t OSA\_TaskDestroy (task\_handler\_t handler)**  
Destroys a previously created task.
- **osa\_status\_t OSA\_TaskYield (void)**  
Puts the active task to the end of scheduler's queue.
- **task\_handler\_t OSA\_TaskGetHandler (void)**  
Gets the handler of active task.
- **uint16\_t OSA\_TaskGetPriority (task\_handler\_t handler)**  
Gets the priority of a task.
- **osa\_status\_t OSA\_TaskSetPriority (task\_handler\_t handler, uint16\_t priority)**  
Sets the priority of a task.

## Message queues

- **msg\_queue\_handler\_t OSA\_MsgQCreate (msg\_queue\_t \*queue, uint16\_t message\_number, uint16\_t message\_size)**

- **osa\_status\_t OSA\_MsgQPut** (*msg\_queue\_handler\_t* handler, void \**pMessage*)  
*Puts a message at the end of the queue.*
- **osa\_status\_t OSA\_MsgQGet** (*msg\_queue\_handler\_t* handler, void \**pMessage*, uint32\_t *timeout*)  
*Reads and removes a message at the head of the queue.*
- **osa\_status\_t OSA\_MsgQDestroy** (*msg\_queue\_handler\_t* handler)  
*Destroys a previously created queue.*

## Memory Management

- **void \* OSA\_MemAlloc** (*size\_t* *size*)  
*Reserves the requested amount of memory in bytes.*
- **void \* OSA\_MemAllocZero** (*size\_t* *size*)  
*Reserves the requested amount of memory in bytes and initializes it to 0.*
- **osa\_status\_t OSA\_MemFree** (void \**ptr*)  
*Releases the memory previously reserved.*

## Time management

- **void OSA\_TimeDelay** (uint32\_t *delay*)  
*Delays execution for a number of milliseconds.*
- **uint32\_t OSA\_TimeGetMsec** (void)  
*Gets the current time since system boot in milliseconds.*

## Interrupt management

- **osa\_status\_t OSA\_InstallIntHandler** (int32\_t *IRQNumber*, void(\**handler*)(void))  
*Installs the interrupt handler.*

## Critical section

- **void OSA\_EnterCritical** (*osa\_critical\_part\_mode\_t* *mode*)  
*Enters the critical part to ensure some code is not preempted.*
- **OSA\_ExitCritical** (*osa\_critical\_part\_mode\_t* *mode*)  
*Exits the critical part.*

## OSA initialize

- **osa\_status\_t OSA\_Init** (void)  
*Initializes the RTOS services.*
- **osa\_status\_t OSA\_Start** (void)  
*Starts the RTOS.*

### 40.0.1 OS Abstraction Layer

#### Overview

Operating System Abstraction layer (OSA) provides a common set of services for drivers and applications so that they can work with or without the operating system. OSA provides services that abstract most of

the OS kernel functionality. These services can either be mapped to the target OS functions directly, or implemented by OSA when no OS is used (bare metal) or when the service does not exist in the target OS. Freescale Kinetis SDK implements the OS abstraction layer for MQX™ RTOS, Free RTOS, μC/OS, and for no OS usage (bare metal). The bare metal OS abstraction implementation is selected as the default option.

OSA provides these services: task management, semaphore, mutex, event, message queue, memory allocator, critical part, and time functions.

## Task Management

With OSA, applications can create and destroy tasks dynamically. These services are mapped to the task functions of RTOSes. For bare metal, a function poll mechanism simulates a task scheduler.

OSA supports task priorities 0~15, where priority 0 is the highest priority and priority 15 is the lowest priority.

To create a task, applications must prepare different resources on different RTOSes. For example, μC-/OS-II and μC/OS-III need pre-allocated task stack while other RTOSes do not need this. The μC/OS-III needs pre-allocated task control block OS\_TCB while other RTOSes do not. To mask the differences, OSA uses a macro OSA\_TASK\_DEFINE to prepare resources for task creation. Then the function [OSA\\_TaskCreate\(\)](#) creates a task based on the resources. This method makes it easy to use a copy of code on different creates a task based on the resources. This method makes it easy to use a copy of code on different RTOSes. However, it is not mandatory to use the OSA\_TASK\_DEFINE. Applications can also prepare the resources manually. There are two methods to create a task:

1. Use the OSA\_TASK\_DEFINE macro and the function [OSA\\_TaskCreate\(\)](#). The macro OSA\_TASK\_DEFINE declares a task handler and task stack statically. The function [OSA\\_TaskCreate\(\)](#) creates task base-on the resources declared by OSA\_TASK\_DEFINE.

This is an example code to create a task using method 1:

```
// Define the task with entry function task_func.
OSA_TASK_DEFINE(task_func, TASK_STACK_SIZE);

void main(void)
{
 task_param_t parameter;

 // Create the task.
 OSA_TaskCreate(task_func, // Task function.
 "my_task", // Task name.
 TASK_STACK_SIZE, // Stack size.
 task_func_stack, // Stack address.
 TASK_PRIO, // Task priority.
 parameter, // Parameter.
 false, // Use float register or not.
 &task_func_task_handler); // Task handler.
 // ...
}
```

2. Prepare resources manually, then use the function [OSA\\_TaskCreate\(\)](#) to create a task.

For example:

```
task_param_t parameter;
task_handler_t task_handler;
```

```

#ifndef FSL_RTOS_UCOSII
 task_stack_t task_stack[TASK_STACK_SIZE/sizeof(task_stack_t)];
 OSA_TaskCreate(task_func, // Task function.
 "my_task", // Task name.
 TASK_STACK_SIZE, // Stack size.
 task_stack, // Stack address.
 TASK_PRIO, // Task priority.
 parameter, // Parameter.
 false, // Use float register or not.
 &task_handler); // Task handler.

```

```

#elif defined(FSL_RTOS_UCOSIII)
 task_stack_t task_stack[TASK_STACK_SIZE/sizeof(task_stack_t)];
 OS_TCB TCB_task;
 task_handler = &TCB_task;
 OSA_TaskCreate(task_func, // Task function.
 "my_task", // Task name.
 TASK_STACK_SIZE, // Stack size.
 task_stack, // Stack address.
 TASK_PRIO, // Task priority.
 parameter, // Parameter.
 false, // Use float register or not.
 &task_handler); // Task handler.

```

```

#else // For MQX, FreeRTOS and bare metal.
 OSA_TaskCreate(task_func, // Task function.
 "my_task", // Task name.
 TASK_STACK_SIZE, // Stack size.
 NULL, // Stack address.
 TASK_PRIO, // Task priority.
 parameter, // Parameter.
 false, // Use float register or not.
 &task_handler); // Task handler.
#endif

```

Method 1 is easy to use. The disadvantage is that one task function can only create one task instance. Method 2 can create multiple task instances using one task function, but the code must be divided by macros for different RTOSes. Applications can choose either method according to requirements.

After a task is created successfully, task handler can be used to manage the task, for example, get or set task priority, destroy task and so on.

If task is not used any more, use the [OSA\\_TaskDestroy\(\)](#) function to destroy the task. If the task function does not contain an infinite loop, or in other words, the task function returns, call the [OSA\\_TaskDestroy\(\)](#) function at the end of the task function.

## Semaphore

The OSA provides the drivers and applications with a counting semaphore. It can be used either to synchronize tasks or to synchronize a task and an ISR.

A semaphore must be initialized with the [OSA\\_SemaCreate\(\)](#) function before using. The semaphore can be initialized with an initial value. When the semaphore is not used any more, use the [OSA\\_SemaDestroy\(\)](#) function to destroy it.

Note that only one task should wait per semaphore. If multiple tasks are waiting per semaphore, different RTOSes may have different behaviors.

This is an example code to create and destroy a semaphore:

```
semaphore_t sem;
```

```
// Initialize the semaphore with initial value.
OSA_SemaCreate(&sem, 0);

// Destroy the semaphore.
OSA_SemaDestroy(&sem);
```

The function [OSA\\_SemaWait\(\)](#) waits a semaphore within the timeout (in milliseconds). Passing a value 0 as a timeout means return immediately and passing the [OSA\\_WAIT\\_FOREVER](#) means wait indefinitely. This function should not be used in the ISR.

The function [OSA\\_SemaPost\(\)](#) wakes up task which is waiting for the semaphore.

## Mutex

A mutex is used for the mutual exclusion of tasks when they access a shared resource. OSA provides a non-recursive mutex, which means a task cannot try to lock a mutex it has already locked.

A mutex must be initialized to an unlocked status with the [OSA\\_MutexCreate\(\)](#) function before using. When the mutex is not used any more, use the [OSA\\_MutexDestroy\(\)](#) function to destroy it.

This is example code to create and destroy a mutex:

```
mutex_t mutex;

// Initialize the mutex
OSA_MutexCreate(&mutex);

// Destroy the mutex
OSA_MutexDestroy(&mutex);
```

The function [OSA\\_MutexLock\(\)](#) waits to lock a mutex within the timeout (in milliseconds). Passing a value 0 as a timeout means return immediately and passing the [OSA\\_WAIT\\_FOREVER](#) means waiting indefinitely.

The function [OSA\\_MutexUnlock\(\)](#) unlocks a mutex which is locked by the current task.

## Event

Event is used for a one-consumer-multi-producer model. It means that there should be only one task waiting for the event. If multiple tasks are waiting on one event, different RTOSes may have different behaviors.

OSA provides two types of events:

1. Auto-clear, which occurs when some task has get flags it is waiting for. These flags are cleared automatically.
2. Manual-clear, which means that the flags could only be cleared manually.

The clear mode is a property of an event. Once an event is created, the clear mode can't be changed.

An event must be initialized with the [OSA\\_EventCreate\(\)](#) function before using. When it is created, its flags are all cleared. When the event is not used any more, use the [event\\_destory\(\)](#) function to destroy it.

This is example code to create and destroy an event object:

```

event_t event;

// Initialize the event
OSA_EventCreate(&event, kEventAutoClear);

// Destroy the event
OSA_EventDestroy(&event);

```

The function [OSA\\_EventWait\(\)](#) waits for specified flags of an event with the timeout (in milliseconds). Passing a value 0 as a timeout means return immediately and passing the OSA\_WAIT\_FOREVER means wait indefinitely. This function can be configured to wait for all specified flags or wait for any one flag in specified flags. The parameter setFlags saves the flags which wake up the waiting task. This function should not be used in the ISR.

The functions [OSA\\_EventSet\(\)](#) and [OSA\\_EventClear\(\)](#) are used to set and clear specified flags of an event.

## Message Queue

OSA provides the FIFO message queue. All messages in a queue have the same size. Message queue holds an internal memory area to save messages. While putting the message, the message entity is copied to this internal memory area. While getting message, message is copied from the internal memory area.

Please note that there should be only one task waiting on one message queue, if multiple tasks are waiting on one message queue, different RTOSes may have different behaviors.

To create a message queue, use [MSG\\_QUEUE\\_DECLARE\(\)](#) and [OSA\\_MsgQCreate\(\)](#) functions as shown here:

```

// Declare the message queue.
MSG_QUEUE_DECLARE(my_message, msg_num, msg_size);

void main(void)
{
 msg_queue_handler_t handler;
 handler = OSA_MsgQCreate(my_message, msg_num, msg_size);
 // ...
}

```

Note that the parameter message\_size is in words and not in bytes. It means that the message queue can only transfer messages of multiple-byte size.

The function [OSA\\_MsgQPut\(\)](#) puts a message to the queue. If the queue is full, an error returns.

The function [OSA\\_MsgQGet\(\)](#) waits for a timeout in milliseconds to get the message from the queue. Passing a value 0 as a timeout means return immediately and passing the OSA\_WAIT\_FOREVER means wait indefinitely. This function should not be used in the ISR.

If the queue is not used any more, use the [OSA\\_MsgQDestroy\(\)](#) function to destroy it.

## Critical Section

OSA provides two types of critical sections. The first type disables the interrupt while the second type only disables the scheduler to stop the task preemption.

## Memory Allocator

The function [OSA\\_MemAlloc\(\)](#) allocates memory with specified size. The function [OSA\\_MemAllocZero\(\)](#) allocates and cleans the memory. The function [OSA\\_MemFree\(\)](#) frees the memory. For RTOSes that have internal memory manager, such as the MQX, OSA maps these functions directly. For other RTOSes or bare metal, the standard functions malloc/calloc/free are used.

## Time Functions

OSA only provides two time functions, [OSA\\_TimeDelay\(\)](#) and [OSA\\_TimeGetMsec\(\)](#). The function [OSA\\_TimeDelay\(\)](#) delays specified time in milliseconds, while the function [OSA\\_TimeGetMsec\(\)](#) gets the system time in milliseconds since POR.

## Interrupt priority

For some RTOSes, a proper interrupt priority must be set if system services are called in this interrupt service routine.

For MQX RTOS, follow these criteria:

1. Interrupt priority must be an even number.
2. Interrupt priority  $\geq 2 * \text{MQX_HARDWARE_INTERRUPT_LEVEL_MAX}$ .

For FreeRTOS, the interrupt priority is defined in the configuration file FreeRTOSConfig.h. In the current configuration, priority 1~15 can be used for the ARM Cortex®-M4. See the FreeRTOS' official documents for details.

## OSA initialization

To initialize and start RTOSes, OSA uses abstract functions [OSA\\_Init\(\)](#) and [OSA\\_Start\(\)](#).

This example shows how to use [OSA\\_Init\(\)](#) and [OSA\\_Start\(\)](#) functions:

```
#include <fsl_os_abstraction.h>

void task_func(task_param_t param)
{
 //...
}

OSA_TASK_DEFINE(task_func, 512);

#if defined(FSL_RTOS_MQX)
void Main_Task(uint32_t param);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
 { 1L, Main_Task, 256L, MQX_MAIN_TASK_PRIORITY, "Main", MQX_AUTO_START_TASK},
 { 0L, 0L, 0L, 0L, 0L, 0L }
};
#endif

#if defined(FSL_RTOS_MQX)
void Main_Task(uint32_t param)
#else
void main(void)
#endif
{
```

```

OSA_Init();
// Other application initialize functions.
// ...
OSA_TaskCreate(task_func, // Task function.
 "my_task", // Task name.
 512, // Stack size.
 task_func_stack, // Stack address.
 5, // Task priority.
 (task_param_t)0, // Parameter.
 false, // Use float register or not.
 &task_func_task_handler); // Task handler.

OSA_Start();
}

```

Note that, for other RTOSes, the task scheduler is started up by the `OSA_Start()` function, but for the MQX RTOS, the task scheduler was started up before the `OSA_Init()` function call. When creating tasks in the `Main_Task()` function with the MQX RTOS, the newly created task runs immediately if it has the highest priority. However, for other RTOSes, all newly created tasks do not run until the `OSA_Start()` function executes.

## 40.1 Enumeration Type Documentation

### 40.1.1 enum osa\_status\_t

Enumerator

- kStatus\_OSA\_Success*** Success.
- kStatus\_OSA\_Error*** Failed.
- kStatus\_OSA\_Timeout*** Timeout occurs while waiting.
- kStatus\_OSA\_Idle*** Used for bare metal only, the wait object is not ready and timeout still not occur.

### 40.1.2 enum osa\_event\_clear\_mode\_t

Enumerator

- kEventAutoClear*** The flags of the event will be cleared automatically.
- kEventManualClear*** The flags of the event will be cleared manually.

### 40.1.3 enum osa\_critical\_section\_mode\_t

Enumerator

- kCriticalSectionSched*** Lock scheduler in critical part.
- kCriticalSectionDisableInt*** Disable interrupt in critical selection.

## Function Documentation

### 40.2 Function Documentation

#### 40.2.1 **osa\_status\_t OSA\_SemaCreate ( semaphore\_t \* pSem, uint8\_t initialValue )**

This function creates a semaphore and sets the value to the parameter initialValue.

## Parameters

|                  |                                             |
|------------------|---------------------------------------------|
| <i>pSem</i>      | Pointer to the semaphore.                   |
| <i>initValue</i> | Initial value the semaphore will be set to. |

## Return values

|                            |                                        |
|----------------------------|----------------------------------------|
| <i>kStatus_OSA_Success</i> | The semaphore is created successfully. |
| <i>kStatus_OSA_Error</i>   | The semaphore can not be created.      |

Example:

```
semaphore_t mySem;
OSA_SemaCreate(&mySem, 0);
```

#### 40.2.2 **osa\_status\_t OSA\_SemaWait ( semaphore\_t \* pSem, uint32\_t timeout )**

This function checks the semaphore's counting value. If it is positive, decreases it and returns kStatus\_O- SA\_Success. Otherwise, a timeout is used to wait.

## Parameters

|                |                                                                                                                                                                            |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pSem</i>    | Pointer to the semaphore.                                                                                                                                                  |
| <i>timeout</i> | The maximum number of milliseconds to wait if semaphore is not positive. Pass OS- A_WAIT_FOREVER to wait indefinitely, pass 0 will return kStatus_OSA_Timeout immediately. |

## Return values

|                            |                                                                                             |
|----------------------------|---------------------------------------------------------------------------------------------|
| <i>kStatus_OSA_Success</i> | The semaphore is received.                                                                  |
| <i>kStatus_OSA_Timeout</i> | The semaphore is not received within the specified 'timeout'.                               |
| <i>kStatus_OSA_Error</i>   | An incorrect parameter was passed.                                                          |
| <i>kStatus_OSA_Idle</i>    | The semaphore is not available and 'timeout' is not exhausted, This is only for bare metal. |

## Note

With bare metal, a semaphore can not be waited by more than one task at the same time.

Example:

```
osa_status_t status;
status = OSA_SemaWait(&mySem, 100);
```

## Function Documentation

```
switch(status)
{
 //...
}
```

### 40.2.3 osa\_status\_t OSA\_SemaPost ( semaphore\_t \* pSem )

Wakes up one task that is waiting on the semaphore. If no task is waiting, increases the semaphore's counting value.

Parameters

|             |                                     |
|-------------|-------------------------------------|
| <i>pSem</i> | Pointer to the semaphore to signal. |
|-------------|-------------------------------------|

Return values

|                            |                                                      |
|----------------------------|------------------------------------------------------|
| <i>kStatus_OSA_Success</i> | The semaphore is successfully signaled.              |
| <i>kStatus_OSA_Error</i>   | The object can not be signaled or invalid parameter. |

Example:

```
osa_status_t status;
status = OSA_SemaPost (&mySem);
switch(status)
{
 //...
}
```

### 40.2.4 osa\_status\_t OSA\_SemaDestroy ( semaphore\_t \* pSem )

Parameters

|             |                                      |
|-------------|--------------------------------------|
| <i>pSem</i> | Pointer to the semaphore to destroy. |
|-------------|--------------------------------------|

Return values

|                            |                                          |
|----------------------------|------------------------------------------|
| <i>kStatus_OSA_Success</i> | The semaphore is successfully destroyed. |
| <i>kStatus_OSA_Error</i>   | The semaphore can not be destroyed.      |

Example:

```
osa_status_t status;
status = OSA_SemaDestroy (&mySem);
switch(status)
{
 //...
}
```

**40.2.5 osa\_status\_t OSA\_MutexCreate ( mutex\_t \* *pMutex* )**

This function creates a non-recursive mutex and sets it to unlocked status.

## Function Documentation

### Parameters

|               |                       |
|---------------|-----------------------|
| <i>pMutex</i> | Pointer to the Mutex. |
|---------------|-----------------------|

### Return values

|                            |                                    |
|----------------------------|------------------------------------|
| <i>kStatus_OSA_Success</i> | The mutex is created successfully. |
| <i>kStatus_OSA_Error</i>   | The mutex can not be created.      |

### Example:

```
mutex_t myMutex;
osa_status_t status;
status = OSA_MutexCreate(&myMutex);
switch (status)
{
 //...
}
```

## 40.2.6 **osa\_status\_t OSA\_MutexLock ( mutex\_t \* *pMutex*, uint32\_t *timeout* )**

This function checks the mutex's status. If it is unlocked, locks it and returns the kStatus\_OSA\_Success. Otherwise, waits for a timeout in milliseconds to lock.

### Parameters

|                |                                                                                                                                                                                                |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pMutex</i>  | Pointer to the Mutex.                                                                                                                                                                          |
| <i>timeout</i> | The maximum number of milliseconds to wait for the mutex. If the mutex is locked, Pass the value OSA_WAIT_FOREVER will wait indefinitely, pass 0 will return k-Status_OSA_Timeout immediately. |

### Return values

|                            |                                                                                         |
|----------------------------|-----------------------------------------------------------------------------------------|
| <i>kStatus_OSA_Success</i> | The mutex is locked successfully.                                                       |
| <i>kStatus_OSA_Timeout</i> | Timeout occurred.                                                                       |
| <i>kStatus_OSA_Error</i>   | Incorrect parameter was passed.                                                         |
| <i>kStatus_OSA_Idle</i>    | The mutex is not available and 'timeout' is not exhausted, This is only for bare metal. |

### Note

This is non-recursive mutex, a task can not try to lock the mutex it has locked.

### Example:

```
osa_status_t status;
status = OSA_MutexLock(&myMutex, 100);
switch (status)
{
 //...
}
```

## 40.2.7 osa\_status\_t OSA\_MutexUnlock ( mutex\_t \* pMutex )

Parameters

|               |                       |
|---------------|-----------------------|
| <i>pMutex</i> | Pointer to the Mutex. |
|---------------|-----------------------|

Return values

|                            |                                                     |
|----------------------------|-----------------------------------------------------|
| <i>kStatus_OSA_Success</i> | The mutex is successfully unlocked.                 |
| <i>kStatus_OSA_Error</i>   | The mutex can not be unlocked or invalid parameter. |

Example:

```
osa_status_t status;
status = OSA_MutexUnlock(&myMutex);
switch (status)
{
 //...
}
```

## 40.2.8 osa\_status\_t OSA\_MutexDestroy ( mutex\_t \* pMutex )

Parameters

|               |                       |
|---------------|-----------------------|
| <i>pMutex</i> | Pointer to the Mutex. |
|---------------|-----------------------|

Return values

|                            |                                      |
|----------------------------|--------------------------------------|
| <i>kStatus_OSA_Success</i> | The mutex is successfully destroyed. |
| <i>kStatus_OSA_Error</i>   | The mutex can not be destroyed.      |

Example:

```
osa_status_t status;
status = OSA_MutexDestroy(&myMutex);
switch (status)
{
 //...
}
```

## Function Documentation

### 40.2.9 osa\_status\_t OSA\_EventCreate ( event\_t \* *pEvent*, osa\_event\_clear\_mode\_t *clearMode* )

This function creates an event object and set its clear mode. If clear mode is kEventAutoClear, when a task gets the event flags, these flags will be cleared automatically. If clear mode is kEventManualClear, these flags must be cleared manually.

#### Parameters

|                  |                                            |
|------------------|--------------------------------------------|
| <i>pEvent</i>    | Pointer to the event object to initialize. |
| <i>clearMode</i> | The event is auto-clear or manual-clear.   |

#### Return values

|                            |                                           |
|----------------------------|-------------------------------------------|
| <i>kStatus_OSA_Success</i> | The event object is successfully created. |
| <i>kStatus_OSA_Error</i>   | The event object is not created.          |

#### Example:

```
event_t myEvent;
OSA_EventCreate(&myEvent, kEventAutoClear);
```

### 40.2.10 osa\_status\_t OSA\_EventWait ( event\_t \* *pEvent*, event\_flags\_t *flagsToWait*, bool *waitForAll*, uint32\_t *timeout*, event\_flags\_t \* *setFlags* )

This function waits for a combination of flags to be set in an event object. Applications can wait for any/all bits to be set. Also this function could obtain the flags who wakeup the waiting task.

#### Parameters

|                    |                                                                                                                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pEvent</i>      | Pointer to the event.                                                                                                                                                                         |
| <i>flagsToWait</i> | Flags that to wait.                                                                                                                                                                           |
| <i>waitForAll</i>  | Wait all flags or any flag to be set.                                                                                                                                                         |
| <i>timeout</i>     | The maximum number of milliseconds to wait for the event. If the wait condition is not met, pass OSA_WAIT_FOREVER will wait indefinitely, pass 0 will return kStatus_OSA_Timeout immediately. |

|                 |                                                                    |
|-----------------|--------------------------------------------------------------------|
| <i>setFlags</i> | Flags that wakeup the waiting task are obtained by this parameter. |
|-----------------|--------------------------------------------------------------------|

Return values

|                            |                                                                                            |
|----------------------------|--------------------------------------------------------------------------------------------|
| <i>kStatus_OSA_Success</i> | The wait condition met and function returns successfully.                                  |
| <i>kStatus_OSA_Timeout</i> | Has not met wait condition within timeout.                                                 |
| <i>kStatus_OSA_Error</i>   | An incorrect parameter was passed.                                                         |
| <i>kStatus_OSA_Idle</i>    | The wait condition is not met and 'timeout' is not exhausted, This is only for bare metal. |

Note

1. With bare metal, a event object can not be waited by more than one tasks at the same time.
  1. Please pay attention to the flags bit width, FreeRTOS uses the most significant 8 bits as control bits, so do not wait these bits while using FreeRTOS.

Example:

```
osa_status_t status;
event_flags_t setFlags;
status = OSA_EventWait(&myEvent, 0x01, true, 100, &setFlags);
switch (status)
{
 //...
}
```

#### 40.2.11 **osa\_status\_t OSA\_EventSet ( event\_t \* *pEvent*, event\_flags\_t *flagsToSet* )**

Sets specified flags of an event object.

Parameters

|                   |                       |
|-------------------|-----------------------|
| <i>pEvent</i>     | Pointer to the event. |
| <i>flagsToSet</i> | Flags to be set.      |

Return values

|                            |                                    |
|----------------------------|------------------------------------|
| <i>kStatus_OSA_Success</i> | The flags were successfully set.   |
| <i>kStatus_OSA_Error</i>   | An incorrect parameter was passed. |

Example:

```
osa_status_t status;
status = OSA_EventSet(&myEvent, 0x01);
switch (status)
{
 //...
}
```

## Function Documentation

### 40.2.12 osa\_status\_t OSA\_EventClear ( *event\_t \* pEvent*, *event\_flags\_t flagsToClear* )

Clears specified flags of an event object.

## Parameters

|                     |                       |
|---------------------|-----------------------|
| <i>pEvent</i>       | Pointer to the event. |
| <i>flagsToClear</i> | Flags to be clear.    |

## Return values

|                            |                                      |
|----------------------------|--------------------------------------|
| <i>kStatus_OSA_Success</i> | The flags were successfully cleared. |
| <i>kStatus_OSA_Error</i>   | An incorrect parameter was passed.   |

Example:

```
osa_status_t status;
status = OSA_EventClear(&myEvent, 0x01);
switch (status)
{
 //...
}
```

**40.2.13 osa\_status\_t OSA\_EventDestroy ( *event\_t \* pEvent* )**

## Parameters

|               |                       |
|---------------|-----------------------|
| <i>pEvent</i> | Pointer to the event. |
|---------------|-----------------------|

## Return values

|                            |                                      |
|----------------------------|--------------------------------------|
| <i>kStatus_OSA_Success</i> | The event is successfully destroyed. |
| <i>kStatus_OSA_Error</i>   | Event destruction failed.            |

Example:

```
osa_status_t status;
status = OSA_EventDestroy(&myEvent);
switch (status)
{
 //...
}
```

**40.2.14 osa\_status\_t OSA\_TaskCreate ( *task\_t task*, *uint8\_t \* name*, *uint16\_t stackSize*, *task\_stack\_t \* stackMem*, *uint16\_t priority*, *task\_param\_t param*, *bool usesFloat*, *task\_handler\_t \* handler* )**

This function is used to create task based on the resources defined by the macro OSA\_TASK\_DEFINE.

## Function Documentation

### Parameters

|                  |                                                      |
|------------------|------------------------------------------------------|
| <i>task</i>      | The task function entry.                             |
| <i>name</i>      | The name of this task.                               |
| <i>stackSize</i> | The stack size in byte.                              |
| <i>stackMem</i>  | Pointer to the stack.                                |
| <i>priority</i>  | Initial priority of the task.                        |
| <i>param</i>     | Pointer to be passed to the task when it is created. |
| <i>usesFloat</i> | This task will use float register or not.            |
| <i>handler</i>   | Pointer to the task handler.                         |

### Return values

|                            |                                   |
|----------------------------|-----------------------------------|
| <i>kStatus_OSA_Success</i> | The task is successfully created. |
| <i>kStatus_OSA_Error</i>   | The task can not be created..     |

Example:

```
osa_status_t status;
OSA_TASK_DEFINE(task_func, stackSize);
status = OSA_TaskCreate(task_func,
 "task_name",
 stackSize,
 task_func_stack,
 prio,
 param,
 false,
 &task_func_task_handler);
switch (status)
{
 //...
}
```

### Note

Use the return value to check whether the task is created successfully. DO NOT check handler. For uC/OS-III, handler is not NULL even if the task creation has failed.

### 40.2.15 osa\_status\_t OSA\_TaskDestroy ( task\_handler\_t *handler* )

## Parameters

|                |                                                                              |
|----------------|------------------------------------------------------------------------------|
| <i>handler</i> | The handler of the task to destroy. Returned by the OSA_TaskCreate function. |
|----------------|------------------------------------------------------------------------------|

## Return values

|                            |                                               |
|----------------------------|-----------------------------------------------|
| <i>kStatus_OSA_Success</i> | The task was successfully destroyed.          |
| <i>kStatus_OSA_Error</i>   | Task destruction failed or invalid parameter. |

## Example:

```
osa_status_t status;
status = OSA_TaskDestroy(myTaskHandler);
switch (status)
{
 //...
}
```

**40.2.16 osa\_status\_t OSA\_TaskYield ( void )**

When a task calls this function, it gives up the CPU and puts itself to the end of a task ready list.

## Return values

|                            |                                      |
|----------------------------|--------------------------------------|
| <i>kStatus_OSA_Success</i> | The function is called successfully. |
| <i>kStatus_OSA_Error</i>   | Error occurs with this function.     |

## Example:

```
osa_status_t status;
status = OSA_TaskYield();
switch (status)
{
 //...
}
```

**40.2.17 task\_handler\_t OSA\_TaskGetHandler ( void )**

## Returns

Handler to current active task.

## Example:

```
task_handler_t handler = OSA_TaskYield();
```

**40.2.18 uint16\_t OSA\_TaskGetPriority ( task\_handler\_t *handler* )**

## Function Documentation

Parameters

|                |                                                     |
|----------------|-----------------------------------------------------|
| <i>handler</i> | The handler of the task whose priority is received. |
|----------------|-----------------------------------------------------|

Returns

Task's priority.

Example:

```
uint16_t taskPrio = OSA_TaskGetPriority(taskHandler);
```

### 40.2.19 osa\_status\_t OSA\_TaskSetPriority ( task\_handler\_t *handler*, uint16\_t *priority* )

Parameters

|                 |                                                     |
|-----------------|-----------------------------------------------------|
| <i>handler</i>  | The handler of the task whose priority is received. |
| <i>priority</i> | The priority to set.                                |

Return values

|                            |                                      |
|----------------------------|--------------------------------------|
| <i>kStatus_OSA_Success</i> | Task's priority is set successfully. |
| <i>kStatus_OSA_Error</i>   | Task's priority can not be set.      |

Example:

```
osa_status_t status;
status = OSA_TaskSetPriority(taskHandler, newPrio);
switch (status)
{
 //...
}
```

### 40.2.20 msg\_queue\_handler\_t OSA\_MsgQCreate ( msg\_queue\_t \* *queue*, uint16\_t *message\_number*, uint16\_t *message\_size* )

This function initializes the message queue that was declared previously. This is an example demonstrating use:

```
msg_queue_handler_t handler;
MSG_QUEUE_DECLARE(my_message, msg_num, msg_size);
handler = OSA_MsgQCreate(my_message, msg_num, msg_size);
```

## Parameters

|                       |                                                         |
|-----------------------|---------------------------------------------------------|
| <i>queue</i>          | The queue declared through the MSG_QUEUE_DECLARE macro. |
| <i>message_number</i> | The number of elements in the queue.                    |
| <i>message_size</i>   | Size of every elements in words.                        |

## Returns

Handler to access the queue for put and get operations. If message queue created failed, return 0.

#### 40.2.21 **osa\_status\_t OSA\_MsgQPut ( msg\_queue\_handler\_t *handler*, void \* *pMessage* )**

This function puts a message to the end of the message queue. If the queue is full, this function returns the kStatus\_OSA\_Error;

## Parameters

|                 |                                                        |
|-----------------|--------------------------------------------------------|
| <i>handler</i>  | Queue handler returned by the OSA_MsgQCreate function. |
| <i>pMessage</i> | Pointer to the message to be put into the queue.       |

## Return values

|                            |                                                        |
|----------------------------|--------------------------------------------------------|
| <i>kStatus_OSA_Success</i> | Message successfully put into the queue.               |
| <i>kStatus_OSA_Error</i>   | The queue was full or an invalid parameter was passed. |

## Example:

```
osa_status_t status;
struct MESSAGE messageToPut = ...;
status = OSA_MsgQPut(queueHandler, &messageToPut);
switch (status)
{
 //...
}
```

#### 40.2.22 **osa\_status\_t OSA\_MsgQGet ( msg\_queue\_handler\_t *handler*, void \* *pMessage*, uint32\_t *timeout* )**

This function gets a message from the head of the message queue. If the queue is empty, timeout is used to wait.

## Function Documentation

### Parameters

|                 |                                                                                                                                                                            |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>handler</i>  | Queue handler returned by the OSA_MsgQCreate function.                                                                                                                     |
| <i>pMessage</i> | Pointer to a memory to save the message.                                                                                                                                   |
| <i>timeout</i>  | The number of milliseconds to wait for a message. If the queue is empty, pass OSA_WAIT_FOREVER will wait indefinitely, pass 0 will return kStatus_OSA_Timeout immediately. |

### Return values

|                            |                                                                                 |
|----------------------------|---------------------------------------------------------------------------------|
| <i>kStatus_OSA_Success</i> | Message successfully obtained from the queue.                                   |
| <i>kStatus_OSA_Timeout</i> | The queue remains empty after timeout.                                          |
| <i>kStatus_OSA_Error</i>   | Invalid parameter.                                                              |
| <i>kStatus_OSA_Idle</i>    | The queue is empty and 'timeout' is not exhausted, This is only for bare metal. |

### Note

With bare metal, there should be only one process waiting on the queue.

### Example:

```
osa_status_t status;
struct MESSAGE messageToGet;
status = OSA_MsgQGet(queueHandler, &messageToGet, 100);
switch (status)
{
 //...
}
```

## 40.2.23 **osa\_status\_t OSA\_MsgQDestroy ( msg\_queue\_handler\_t *handler* )**

### Parameters

|                |                                                        |
|----------------|--------------------------------------------------------|
| <i>handler</i> | Queue handler returned by the OSA_MsgQCreate function. |
|----------------|--------------------------------------------------------|

### Return values

|                            |                                       |
|----------------------------|---------------------------------------|
| <i>kStatus_OSA_Success</i> | The queue was successfully destroyed. |
| <i>kStatus_OSA_Error</i>   | Message queue destruction failed.     |

Example:

```
osa_status_t status;
status = OSA_MsgQDestroy(queueHandler);
switch (status)
{
 //...
}
```

#### 40.2.24 void\* OSA\_MemAlloc ( size\_t size )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>size</i> | Amount of bytes to reserve. |
|-------------|-----------------------------|

Returns

Pointer to the reserved memory. NULL if memory could not be allocated.

#### 40.2.25 void\* OSA\_MemAllocZero ( size\_t size )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>size</i> | Amount of bytes to reserve. |
|-------------|-----------------------------|

Returns

Pointer to the reserved memory. NULL if memory could not be allocated.

#### 40.2.26 osa\_status\_t OSA\_MemFree ( void \* ptr )

Parameters

|            |                                                               |
|------------|---------------------------------------------------------------|
| <i>ptr</i> | Pointer to the start of the memory block previously reserved. |
|------------|---------------------------------------------------------------|

## Function Documentation

Return values

|                            |                                      |
|----------------------------|--------------------------------------|
| <i>kStatus_OSA_Success</i> | Memory correctly freed.              |
| <i>kStatus_OSA_Error</i>   | Error occurs during free the memory. |

### 40.2.27 void OSA\_TimeDelay ( uint32\_t *delay* )

Parameters

|              |                                   |
|--------------|-----------------------------------|
| <i>delay</i> | The time in milliseconds to wait. |
|--------------|-----------------------------------|

### 40.2.28 uint32\_t OSA\_TimeGetMsec ( void )

Returns

Current time in milliseconds.

### 40.2.29 osa\_status\_t OSA\_InstallIntHandler ( int32\_t *IRQNumber*, void(\*)(void) *handler* )

Parameters

|                  |                                   |
|------------------|-----------------------------------|
| <i>IRQNumber</i> | IRQ number of the interrupt.      |
| <i>handler</i>   | The interrupt handler to install. |

Return values

|                            |                                    |
|----------------------------|------------------------------------|
| <i>kStatus_OSA_Success</i> | Handler is installed successfully. |
| <i>kStatus_OSA_Error</i>   | Handler could not be installed.    |

### 40.2.30 void OSA\_EnterCritical ( osa\_critical\_part\_mode\_t *mode* )

Parameters

|             |                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>mode</i> | Lock task scheduler or disable interrupt in critical part. Pass kCriticalLockSched to lock task scheduler, pass kCriticalDisableInt to disable interrupt. |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|

#### 40.2.31 void OSA\_ExitCritical ( osa\_critical\_part\_mode\_t *mode* )

Parameters

|             |                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>mode</i> | Lock task scheduler or disable interrupt in critical part. Pass kCriticalLockSched to lock task scheduler, pass kCriticalDisableInt to disable interrupt. |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|

#### 40.2.32 osa\_status\_t OSA\_Init ( void )

This function sets up the basic RTOS services. It should be called first in the main function.

Return values

|                            |                                             |
|----------------------------|---------------------------------------------|
| <i>kStatus_OSA_Success</i> | RTOS services are initialized successfully. |
| <i>kStatus_OSA_Error</i>   | Error occurs during initialization.         |

#### 40.2.33 osa\_status\_t OSA\_Start ( void )

This function starts the RTOS scheduler and may never return.

Return values

|                            |                                  |
|----------------------------|----------------------------------|
| <i>kStatus_OSA_Success</i> | RTOS starts to run successfully. |
| <i>kStatus_OSA_Error</i>   | Error occurs when start RTOS.    |

## Bare Metal Abstraction Layer

### 40.3 Bare Metal Abstraction Layer

The Kinetis SDK provides the Bare Metal Abstraction Layer for synchronization, mutual exclusion, message queue, etc.

#### Data Structures

- struct [semaphore\\_t](#)  
*Type for an semaphore. [More...](#)*
- struct [mutex\\_t](#)  
*Type for a mutex. [More...](#)*
- struct [event\\_t](#)  
*Type for an event object. [More...](#)*
- struct [task\\_control\\_block\\_t](#)  
*Task control block for bare metal. [More...](#)*
- struct [msg\\_queue\\_t](#)  
*Type for a message queue. [More...](#)*

#### Macros

- #define [OSA\\_WAIT\\_FOREVER](#) 0xFFFFFFFFU  
*Constant to pass as timeout value in order to wait indefinitely.*
- #define [TASK\\_MAX\\_NUM](#) 5  
*How many tasks can the bare metal support.*

#### Typedefs

- typedef uint32\_t [event\\_flags\\_t](#)  
*Type for an event flags group, bit 32 is reserved.*
- typedef void \* [task\\_param\\_t](#)  
*Type for task parameter.*
- typedef void(\* [task\\_t](#))([task\\_param\\_t](#) param)  
*Type for a task pointer.*
- typedef [task\\_control\\_block\\_t](#) \* [task\\_handler\\_t](#)  
*Type for a task handler, returned by the OSA\_TaskCreate function.*
- typedef uint32\_t [task\\_stack\\_t](#)  
*Type for a task stack.*
- typedef [msg\\_queue\\_t](#) \* [msg\\_queue\\_handler\\_t](#)  
*Type for a message queue handler.*

#### Thread management

- void [OSA\\_PollAllOtherTasks](#) (void)  
*Calls all task functions one time except for the current task.*
- #define [OSA\\_TASK\\_DEFINE](#)(task, stackSize)  
*Defines a task.*

## Message queues

- #define **MSG\_QUEUE\_DECLARE**(name, number, size)  
*This macro statically reserves the memory required for the queue.*

### 40.3.0.1 Bare Metal Abstraction Layer

#### Overview

When RTOSes are not used, bare metal abstraction layer provides semaphore, mutex, event, message queue and so on. Because bare metal does not have a task scheduler, it is necessary to exercise caution while using bare metal abstraction layer.

#### Bare Metal's Task Management

By contrast to RTOSes, bare metal abstraction layer uses a poll mechanism to simulate a task. All task functions are linked into a list and called one by one. Therefore, bare metal task function should not contain an infinite loop. It must return at a proper time to let the other tasks run.

Bare metal task does not support priority, all tasks use the same priority. The macro **TASK\_MAX\_NUM** defines how many tasks applications could create. If it is set to 0, then applications could not use task APIs.

#### Bare Metal's Wait Functions

Bare metal wait functions, such as **OSA\_SemaWait** and **OSA\_EventWait**, return the **kStatus\_OSA\_Idle** if wait condition is not met and timeout has not occurred. Applications should catch this value and take proper actions. If the wait condition is set by the ISR, applications could wait in a loop:

```
void post_ISR(void)
{
 //...
 OSA_SemaPost (&my_sem);
 //...
}

void wait_task(task_param_t param)
{
 //...
 do
 {
 status = OSA_SemaWait (&my_sem, 10);
 } while(kStatus_OSA_Idle == status);

 //...
}
```

## Bare Metal Abstraction Layer

In this example, if my\_sem is not posted by post\_ISR, but posted by a task post\_task, then OSA\_SemaWait in loop could never get my\_sem within timeout, because post\_task does not have chance to post my\_sem. In this situation, applications could be implemented like this:

```
void post_task(task_param_t param)
{
 //...
 OSA_SemaPost(&my_sem);
 //...
}

void wait_task(task_param_t param)
{
 status = OSA_SemaWait(&my_sem, 10);

 switch (status)
 {
 case kStatus_OSA_Idle:
 return;
 case kStatus_OSA_Success:
 // ...
 break;
 case kStatus_OSA_Error:
 // ...
 break;
 case kStatus_OSA_Timeout:
 // ...
 break;
 }

 //...
}
```

Wait the semaphore at the start of the task, if kStatus\_OSA\_Idle is got, return and let other task run, then post\_task has chance to post my\_sem.

The other method is using function [OSA\\_PollAllOtherTasks\(\)](#). This function calls all other tasks one time, then post\_task could post my\_sem.

```
void post_task(task_param_t param)
{
 //...
 OSA_SemaPost(&my_sem);
 //...
}

void wait_task(task_param_t param)
{
 //...
 status = OSA_SemaWait(&my_sem, 10);

 while (kStatus_OSA_Idle == status)
 {
 OSA_PollAllOtherTasks();
 status = OSA_SemaWait(&my_sem, 10);
 }

 //...
}
```

The limitation of this method is that only one task can use the [OSA\\_PollAllOtherTasks\(\)](#) function. If both task\_A and task\_B call this function, then the stack overflow may occur, because the call stack is like this: task\_A -> OSA\_PollAllOtherTasks -> task\_B -> OSA\_PollAllOtherTasks -> task\_A -> ...

## Bare Metal Time management

Bare metal OSA maintains a system time by the lowpower timer module. Applications can get system time in milliseconds using the function [OSA\\_TimeGetMsec\(\)](#). At the same time, all wait functions such as [OSA\\_SemaWait\(\)](#), [OSA\\_EventWait\(\)](#) depend on this system time. Lowpower timer module is set up in the function [OSA\\_Init\(\)](#). To use this time function, ensure that the [OSA\\_Init\(\)](#) function is called. Please note that lowpower timer provides only 16-bit time count, it wraps every 65536ms. So take care to use these three kinds of functions:

1. [OSA\\_TimeDelay\(\)](#) can not delay longer than 65536ms.
2. Wait functions, such as [OSA\\_SemaWait\(\)](#), can not set timeout longer than 65536ms, however, OS-A\_WAIT\_FOREVER is allowed.
3. [OSA\\_TimeGetMsec\(\)](#) wraps every 65536ms, if it does not meet the requirement, please implement use other timer module in application.

### 40.3.1 Data Structure Documentation

#### 40.3.1.1 struct semaphore\_t

##### Data Fields

- volatile bool [isWaiting](#)  
*Is any task waiting for a timeout on this object.*
- volatile uint8\_t [semCount](#)  
*The count value of the object.*
- uint32\_t [time\\_start](#)  
*The time to start timeout.*
- uint32\_t [timeout](#)  
*Timeout to wait in milliseconds.*

#### 40.3.1.2 struct mutex\_t

##### Data Fields

- volatile bool [isWaiting](#)  
*Is any task waiting for a timeout on this mutex.*
- volatile bool [isLocked](#)  
*Is the object locked or not.*
- uint32\_t [time\\_start](#)  
*The time to start timeout.*
- uint32\_t [timeout](#)  
*Timeout to wait in milliseconds.*

### 40.3.1.3 struct event\_t

#### Data Fields

- volatile bool `isWaiting`  
*Is any task waiting for a timeout on this event.*
- `uint32_t time_start`  
*The time to start timeout.*
- `uint32_t timeout`  
*Timeout to wait in milliseconds.*
- volatile `event_flags_t flags`  
*The flags status.*
- `osa_event_clear_mode_t clearMode`  
*Auto clear or manual clear.*

### 40.3.1.4 struct task\_control\_block\_t

#### Data Fields

- `task_t p_func`  
*Task's entry.*
- `task_param_t param`  
*Task's parameter.*
- `struct TaskControlBlock * next`  
*Pointer to next task control block.*
- `struct TaskControlBlock * prev`  
*Pointer to previous task control block.*

### 40.3.1.5 struct msg\_queue\_t

#### Data Fields

- `uint32_t * queueMem`  
*Points to the queue memory.*
- `uint16_t number`  
*The number of messages in the queue.*
- `uint16_t size`  
*The size in words of each message.*
- `uint16_t head`  
*Index of the next message to be read.*
- `uint16_t tail`  
*Index of the next place to write to.*
- `semaphore_t queueSem`  
*Semaphore wakeup tasks waiting for msg.*
- volatile bool `isEmpty`  
*Whether queue is empty.*

### 40.3.2 Macro Definition Documentation

#### 40.3.2.1 #define OSA\_WAIT\_FOREVER 0xFFFFFFFFU

#### 40.3.2.2 #define TASK\_MAX\_NUM 5

#### 40.3.2.3 #define OSA\_TASK\_DEFINE( task, stackSize )

**Value:**

```
task_stack_t* task##_stack = NULL; \
 task_handler_t task##_task_handler
```

This macro defines resources for a task statically. Then, the OSA\_TaskCreate creates the task based-on these resources.

Parameters

|                  |                                          |
|------------------|------------------------------------------|
| <i>task</i>      | The task function.                       |
| <i>stackSize</i> | The stack size this task needs in bytes. |

#### 40.3.2.4 #define MSG\_QUEUE\_DECLARE( name, number, size )

**Value:**

```
uint32_t queueMem_##name[number * size]; \
 msg_queue_t
entity_##name = { \
 .queueMem = queueMem_##name \
}; \
msg_queue_t
*name = &(entity_##name)
```

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>name</i>   | Identifier for the memory region. |
| <i>number</i> | Number of elements in the queue.  |
| <i>size</i>   | Size of every element in words.   |

### 40.3.3 Function Documentation

#### 40.3.3.1 void OSA\_PollAllOtherTasks ( void )

This function calls all other task functions one time. If current task is waiting for an event triggered by other tasks, this function could be used to trigger the event.

## Bare Metal Abstraction Layer

### Note

There should be only one task calls this function, if more than one task call this function, stack overflow may occurs. Be careful to use this function.

## 40.4 MQX Abstraction Layer

The Kinetis SDK provides the MQX Abstraction Layer for synchronization, mutual exclusion, message queue, etc.

### Macros

- `#define OSA_WAIT_FOREVER 0xFFFFFFFFFU`  
*Constant to pass as timeout value in order to wait indefinitely.*
- `#define MQX_MAIN_TASK_PRIORITY (7+16)`  
*The priority of MQX's Main\_Task.*

### Typedefs

- `typedef LWSEM_STRUCT semaphore_t`  
*Type for a semaphore.*
- `typedef MUTEX_STRUCT mutex_t`  
*Type for a mutex.*
- `typedef LWEVENT_STRUCT event_t`  
*Type for an event group object.*
- `typedef _mqx_uint event_flags_t`  
*Type for an event flags group, bit 32 is reserved.*
- `typedef TASK_FPTR task_t`  
*Type for a task pointer.*
- `typedef _task_id task_handler_t`  
*Type for a task handler, returned by the OSA\_TaskCreate function.*
- `typedef uint32_t task_param_t`  
*Type for a task handler, returned by the task\_create function.*
- `typedef uint32_t task_stack_t`  
*Type for a task stack.*
- `typedef _mqx_max_type msg_queue_t`  
*Type for a message queue.*
- `typedef void * msg_queue_handler_t`  
*Type for a message queue handler.*

### Thread management

- `#define OSA_TASK_DEFINE(task, stackSize)`  
*Defines a task.*
- `#define PRIORITY_OSA_TORTOS(osa_prio) ((osa_prio)+7U)`  
*To provide unified task priority for upper layer, OSA layer makes conversion.*
- `#define PRIORITY_RTOS_TOOSA(rtos_prio) ((rtos_prio)-7U)`

### Message queues

- `void * OSA_MemoryAllocateAlign (size_t size, size_t align)`

## MQX Abstraction Layer

*Allocates the block aligned at a specific boundary.*

- `#define SIZE_IN_MMT_UNITS(size) ((size + sizeof(_mqx_max_type) - 1) / sizeof(_mqx_max_type))`
- `#define MSG_QUEUE_DECLARE(name, number, size) _mqx_max_type name[SIZE_IN_MMT_UNITS(sizeof(LWMSGQ_STRUCT)) + SIZE_IN_MMT_UNITS(size * 4) * number]`

*This macro statically reserves the memory required for the queue.*

### 40.4.1 Macro Definition Documentation

#### 40.4.1.1 #define OSA\_WAIT\_FOREVER 0xFFFFFFFFU

#### 40.4.1.2 #define MQX\_MAIN\_TASK\_PRIORITY (7+16)

#### 40.4.1.3 #define OSA\_TASK\_DEFINE( task, stackSize )

**Value:**

```
task_stack_t* task##_stack = NULL; \
task_handler_t task##_task_handler
```

This macro defines resources for a task statically. Then, the OSA\_TaskCreate creates the task based-on these resources.

Parameters

|                  |                                          |
|------------------|------------------------------------------|
| <i>task</i>      | The task function.                       |
| <i>stackSize</i> | The stack size this task needs in bytes. |

#### 40.4.1.4 #define PRIORITY\_OSA\_TORTOS( osa\_prio ) ((osa\_prio)+7U)

MQX's highest 7 priorities are special priorities.

#### 40.4.1.5 #define MSG\_QUEUE\_DECLARE( name, number, size ) \_mqx\_max\_type name[SIZE\_IN\_MMT\_UNITS(sizeof(LWMSGQ\_STRUCT)) + SIZE\_IN\_MMT\_UNITS(size \* 4) \* number]

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>name</i>   | Identifier for the memory region. |
| <i>number</i> | Number of elements in the queue.  |
| <i>size</i>   | Size of element in 4B units.      |

## 40.4.2 Typedef Documentation

40.4.2.1 **typedef LWSEM\_STRUCT semaphore\_t**

40.4.2.2 **typedef MUTEX\_STRUCT mutex\_t**

40.4.2.3 **typedef LWEVENT\_STRUCT event\_t**

40.4.2.4 **typedef \_mqx\_uint event\_flags\_t**

40.4.2.5 **typedef TASK\_FPTR task\_t**

40.4.2.6 **typedef \_task\_id task\_handler\_t**

40.4.2.7 **typedef uint32\_t task\_param\_t**

40.4.2.8 **typedef uint32\_t task\_stack\_t**

40.4.2.9 **typedef \_mqx\_max\_type msg\_queue\_t**

40.4.2.10 **typedef void\* msg\_queue\_handler\_t**

### 40.5 μC/OS-II Abstraction Layer

The Kinetis SDK provides the μC/OS-II Abstraction Layer for synchronization, mutual exclusion, message queue, etc.

#### Data Structures

- struct [event\\_ucosii](#)  
*Type for an event group object in μCOS-II. [More...](#)*
- struct [msgq\\_ucosii](#)  
*Type for message queue in μCOS-II. [More...](#)*

#### Macros

- #define [OSA\\_WAIT\\_FOREVER](#) 0xFFFFFFFFU  
*Constant to pass as timeout value to wait indefinitely.*

#### Typedefs

- typedef OS\_FLAGS [event\\_flags\\_t](#)  
*Type for an event flags group.*
- typedef OS\_EVENT \* [semaphore\\_t](#)  
*Type for an semaphore.*
- typedef OS\_EVENT \* [mutex\\_t](#)  
*Type for a mutex.*
- typedef [event\\_ucosii](#) [event\\_t](#)  
*Type for an event group object.*
- typedef [msgq\\_ucosii](#) [msg\\_queue\\_t](#)  
*Type for a message queue.*
- typedef [msgq\\_ucosii](#) \* [msg\\_queue\\_handler\\_t](#)  
*Type for a message queue handler.*
- typedef OS\_TCB \* [task\\_handler\\_t](#)  
*Type for a task handler, returned by the OSA\_TaskCreate function.*
- typedef OS\_STK [task\\_stack\\_t](#)  
*Type for a task stack.*
- typedef void \* [task\\_param\\_t](#)  
*Type for task parameter.*
- typedef void(\* [task\\_t](#))([task\\_param\\_t](#) param)  
*Type for a task pointer.*

#### Thread management

- #define [OSA\\_TASK\\_DEFINE](#)(task, stackSize)  
*Defines a task.*
- #define [PRIORITY\\_OSA\\_TORTOS](#)(osa\_prio) ((osa\_prio)+4U)  
*To provide unified task priority for upper layer, OSA layer makes conversion.*

- #define **PRIORITY\_RTO\_SOA**(rtos\_prio) ((rtos\_prio)-4U)

## Message queues

- #define **MSG\_QUEUE\_DECLARE**(name, number, size)  
*This macro statically reserves the memory required for the queue.*

### 40.5.1 Data Structure Documentation

#### 40.5.1.1 struct event\_ucosii

##### Data Fields

- OS\_FLAG\_GRP \* pGroup  
*Pointer to μCOS-II's event entity.*
- osa\_event\_clear\_mode\_t clearMode  
*Auto clear or manual clear.*

#### 40.5.1.2 struct msgq\_ucosii

##### Data Fields

- OS\_EVENT \* pQueue  
*Pointer to the queue.*
- void \*\* msgTbl  
*Pointer to the array that saves the pointers to messages.*
- OS\_MEM \* pMem  
*Pointer to memory where save the messages.*
- void \* msgs  
*Memory to save the messages.*
- uint16\_t size  
*Size of the message in words.*

### 40.5.2 Macro Definition Documentation

#### 40.5.2.1 #define OSA\_WAIT\_FOREVER 0xFFFFFFFFU

#### 40.5.2.2 #define OSA\_TASK\_DEFINE( task, stackSize )

##### Value:

```
task_stack_t task##_stack[(stackSize)/sizeof(task_stack_t)];
\
task_handler_t task##_task_handler
```

## **µC/OS-II Abstraction Layer**

This macro defines resources for a task statically. Then the OSA\_TaskCreate creates the task based on these resources.

## Parameters

|                  |                                          |
|------------------|------------------------------------------|
| <i>task</i>      | The task function.                       |
| <i>stackSize</i> | The stack size this task needs in bytes. |

**40.5.2.3 #define MSG\_QUEUE\_DECLARE( *name*, *number*, *size* )****Value:**

```
void* msgTbl_##name[number];
 uint32_t msgs_##name[number*size];
msg_queue_t memory_##name = {
 .msgTbl = msgTbl_##name,
 .msgs = msgs_##name
};
msg_queue_t *name = &(memory_##name)
```

## Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>name</i>   | Identifier for the memory region. |
| <i>number</i> | Number of elements in the queue.  |
| <i>size</i>   | Size of every elements in words.  |

### 40.6 μC/OS-III Abstraction Layer

The Kinetis SDK provides the μC/OS-III Abstraction Layer for synchronization, mutual exclusion, message queue, etc.

#### Data Structures

- struct [event\\_ucosiii](#)  
*Type for an event group object in μCOS-III. [More...](#)*
- struct [msgq\\_struct\\_ucosiii](#)  
*Type for message queue in μCOS-III. [More...](#)*

#### Macros

- #define [OSA\\_WAIT\\_FOREVER](#) 0xFFFFFFFFU  
*Constant to pass as timeout value in order to wait indefinitely.*

#### Typedefs

- typedef OS\_TCB \* [task\\_handler\\_t](#)  
*Type for a task handler, returned by the OSA\_TaskCreate function.*
- typedef CPU\_STK [task\\_stack\\_t](#)  
*Type for a task stack.*
- typedef void \* [task\\_param\\_t](#)  
*Type for task parameter.*
- typedef void(\* [task\\_t](#))([task\\_param\\_t](#) param)  
*Type for a task pointer.*
- typedef OS\_SEM [semaphore\\_t](#)  
*Type for a semaphore.*
- typedef OS\_MUTEX [mutex\\_t](#)  
*Type for a mutex.*
- typedef OS\_FLAGS [event\\_flags\\_t](#)  
*Type for an event flags group.*
- typedef [event\\_ucosiii](#) [event\\_t](#)  
*Type for an event object.*
- typedef [msgq\\_struct\\_ucosiii](#) [msg\\_queue\\_t](#)  
*Type for a message queue.*
- typedef [msg\\_queue\\_t](#) \* [msg\\_queue\\_handler\\_t](#)  
*Type for a message queue handler.*

#### Thread management

- #define [OSA\\_TASK\\_DEFINE](#)(task, stackSize)  
*Defines a task.*
- #define [PRIORITY\\_OSA\\_TORTOS](#)(osa\_prio) ((osa\_prio)+1U)  
*To provide unified task priority for upper layer, OSA layer makes conversion.*

- #define **PRIORITY\_RTOs\_TO\_OSA**(rtos\_prio) ((rtos\_prio)-1U)

## Message queues

- #define **MSG\_QUEUE\_DECLARE**(name, number, size)  
*This macro statically reserves the memory required for the queue.*

### 40.6.1 Data Structure Documentation

#### 40.6.1.1 struct event\_ucosiii

##### Data Fields

- OS\_FLAG\_GRP **group**  
*μCOS-III's event entity.*
- **osa\_event\_clear\_mode\_t clearMode**  
*Auto clear or manual clear.*

#### 40.6.1.2 struct msgq\_struct\_ucosiii

##### Data Fields

- OS\_Q **queue**  
*the message queue's control block*
- OS\_MEM **mem**  
*control block for the memory where save the messages*
- void \* **msgs**  
*pointer to the memory where save the messages*
- uint16\_t **size**  
*size of the message in words*

### 40.6.2 Macro Definition Documentation

#### 40.6.2.1 #define OSA\_WAIT\_FOREVER 0xFFFFFFFFU

#### 40.6.2.2 #define OSA\_TASK\_DEFINE( task, stackSize )

##### Value:

```
OS_TCB TCB_##task;
task_stack_t task##_stack[(stackSize)/sizeof(task_stack_t)];
task_handler_t task##_task_handler = &(TCB_##task)
```

This macro defines resources for a task statically. Then, the OSA\_TaskCreate creates the task based on these resources.

## µC/OS-III Abstraction Layer

Parameters

|                  |                                          |
|------------------|------------------------------------------|
| <i>task</i>      | The task function.                       |
| <i>stackSize</i> | The stack size this task needs in bytes. |

### 40.6.2.3 #define PRIORITY\_OSA\_TO\_RTOS( *osa\_prio* ) ((*osa\_prio*)+1U)

uC/OS-III's tick task should have a high priority, so we set tick task to priority 0, applications use other priorities.

### 40.6.2.4 #define MSG\_QUEUE\_DECLARE( *name*, *number*, *size* )

**Value:**

```
uint32_t msgs_##name[number*size];
msg_queue_t memory_##name = {
 .msgs = msgs_##name
};
msg_queue_t *name = &(memory_##name)
```

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>name</i>   | Identifier for the memory region. |
| <i>number</i> | Number of elements in the queue.  |
| <i>size</i>   | Size of every elements in words.  |

### 40.6.3 Typedef Documentation

40.6.3.1 **typedef OS\_TCB\* task\_handler\_t**

40.6.3.2 **typedef CPU\_STK task\_stack\_t**

40.6.3.3 **typedef OS\_SEM semaphore\_t**

40.6.3.4 **typedef OS\_MUTEX mutex\_t**

40.6.3.5 **typedef OS\_FLAGS event\_flags\_t**

40.6.3.6 **typedef event\_ucosiii event\_t**

40.6.3.7 **typedef msgq\_struct\_ucosiii msg\_queue\_t**

40.6.3.8 **typedef msg\_queue\_t\* msg\_queue\_handler\_t**

## FreeRTOS Abstraction Layer

### 40.7 FreeRTOS Abstraction Layer

The Kinetis SDK provides the FreeRTOS Abstraction Layer for synchronization, mutual exclusion, message queue, etc.

#### Data Structures

- struct `event_freertos`  
*Type for an event group object in FreeRTOS. More...*

#### Macros

- #define `OSA_WAIT_FOREVER` 0xFFFFFFFFU  
*Constant to pass as timeout value in order to wait indefinitely.*

#### Typedefs

- typedef TaskHandle\_t `task_handler_t`  
*Type for a task handler, returned by the OSA\_TaskCreate function.*
- typedef portSTACK\_TYPE `task_stack_t`  
*Type for a task stack.*
- typedef void \* `task_param_t`  
*Type for task parameter.*
- typedef pdTASK\_CODE `task_t`  
*Type for a task function.*
- typedef xSemaphoreHandle `mutex_t`  
*Type for a mutex.*
- typedef xSemaphoreHandle `semaphore_t`  
*Type for a semaphore.*
- typedef EventBits\_t `event_flags_t`  
*Type for an event flags object.*
- typedef `event_freertos event_t`  
*Type for an event group object.*
- typedef xQueueHandle `msg_queue_t`  
*Type for a message queue declaration and creation.*
- typedef xQueueHandle `msg_queue_handler_t`  
*Type for a message queue handler.*

#### Thread management

- #define `OSA_TASK_DEFINE(task, stackSize)`  
*Creates a task descriptor that is used to create the task with OSA\_TaskCreate.*
- #define `PRIORITY_OSA_TO_RTOs(osa_prio) (configMAX_PRIORITIES - (osa_prio) -2)`  
*To provide unified task priority for upper layer, OSA layer makes conversion.*
- #define `PRIORITY_RTOs_TO_OSA(rtos_prio) (configMAX_PRIORITIES - (rtos_prio) -2)`

## Message queues

- #define **MSG\_QUEUE\_DECLARE**(name, number, size) **msg\_queue\_t** \*name = NULL  
*This macro statically reserves the memory required for the queue.*

### 40.7.1 Data Structure Documentation

#### 40.7.1.1 struct event\_freertos

##### Data Fields

- EventGroupHandle\_t **eventHandler**  
*FreeRTOS event handler.*
- osa\_event\_clear\_mode\_t **clearMode**  
*Auto clear or manual clear.*

### 40.7.2 Macro Definition Documentation

#### 40.7.2.1 #define OSA\_WAIT\_FOREVER 0xFFFFFFFFU

#### 40.7.2.2 #define OSA\_TASK\_DEFINE( task, stackSize )

##### Value:

```
task_stack_t* task##_stack = NULL; \
 task_handler_t task##_task_handler
```

##### Parameters

|                  |                                                |
|------------------|------------------------------------------------|
| <b>task</b>      | The task function.                             |
| <b>stackSize</b> | Number of elements in the stack for this task. |

#### 40.7.2.3 #define MSG\_QUEUE\_DECLARE( name, number, size ) msg\_queue\_t \*name = NULL

##### Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>name</i>   | Identifier for the memory region. |
| <i>number</i> | Number of elements in the queue.  |
| <i>size</i>   | Size of every elements in words.  |

### 40.7.3 Typedef Documentation

40.7.3.1 **typedef TaskHandle\_t task\_handler\_t**

40.7.3.2 **typedef portSTACK\_TYPE task\_stack\_t**

40.7.3.3 **typedef pdTASK\_CODE task\_t**

40.7.3.4 **typedef xSemaphoreHandle mutex\_t**

40.7.3.5 **typedef xSemaphoreHandle semaphore\_t**

40.7.3.6 **typedef EventBits\_t event\_flags\_t**

40.7.3.7 **typedef xQueueHandle msg\_queue\_t**

# Chapter 41

## Hwtimer\_driver

### Data Structures

- struct [hwtimer\\_ptr\\_t](#)  
*Hwtimer structure.* [More...](#)
- struct [hwtimer\\_time\\_ptr\\_t](#)  
*Hwtimer\_time structure.* [More...](#)
- struct [hwtimer\\_devif\\_ptr\\_t](#)  
*hwtimer\_devif structure.* [More...](#)

### Macros

- #define [HWTIMER\\_LL\\_CONTEXT\\_LEN](#) 5U  
*Hwtimer low level context data length definition.*

### Typedefs

- typedef void(\* [hwtimer\\_callback\\_t](#) )(void \*p)  
*Define for low level context data length.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_init\\_t](#) )(hwtimer\_t \*hwtimer, uint32\_t id, uint32\_t isr\_prior, void \*data)  
*Type defines init function for devif structure.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_deinit\\_t](#) )(hwtimer\_t \*hwtimer)  
*Type defines deinit function for devif structure.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_set\\_div\\_t](#) )(hwtimer\_t \*hwtimer, uint32\_t divider)  
*Type defines set\_div function for devif structure.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_start\\_t](#) )(hwtimer\_t \*hwtimer)  
*Type defines start function for devif structure.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_stop\\_t](#) )(hwtimer\_t \*hwtimer)  
*Type defines stop function for devif structure.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_reset\\_t](#) )(hwtimer\_t \*hwtimer)  
*Type defines reset function for devif structure.*
- typedef [\\_hwtimer\\_error\\_code\\_t](#)(\* [hwtimer\\_devif\\_get\\_time\\_t](#) )(hwtimer\_t \*hwtimer, [hwtimer\\_time\\_t](#) \*time)  
*Type defines get\_time function for devif structure.*

## Enumerations

- enum `_hwtimer_error_code_t` {  
  `kHwtimerSuccess`,  
  `kHwtimerInvalidInput`,  
  `kHwtimerInvalidPointer`,  
  `kHwtimerClockManagerError`,  
  `kHwtimerRegisterHandlerError`,  
  `kHwtimerLockError`,  
  `kHwtimerUnknown` }

*Hwtimer error codes definition.*

## Functions

- `_hwtimer_error_code_t HWTIMER_SYS_Init (hwtimer_t *hwtimer, const hwtimer_devif_t *k-Devif, uint32_t id, uint32_t isrPrior, void *data)`  
*Initializes caller allocated structure according to given parameters.*
- `_hwtimer_error_code_t HWTIMER_SYS_Deinit (hwtimer_t *hwtimer)`  
*De-initialization of the hwtimer.*
- `_hwtimer_error_code_t HWTIMER_SYS_SetFreq (hwtimer_t *hwtimer, clock_names_t clock-Name, uint32_t freq)`  
*The function configures timer to tick at frequency as close as possible to the requested one.*
- `_hwtimer_error_code_t HWTIMER_SYS_SetPeriod (hwtimer_t *hwtimer, clock_names_t clock-Name, uint32_t period)`  
*Set period of hwtimer.*
- `uint32_t HWTIMER_SYS_GetFreq (hwtimer_t *hwtimer)`  
*Get frequency of hwtimer.*
- `uint32_t HWTIMER_SYS_GetPeriod (hwtimer_t *hwtimer)`  
*Get period of hwtimer.*
- `_hwtimer_error_code_t HWTIMER_SYS_Start (hwtimer_t *hwtimer)`  
*Enables the timer and leaves it running.*
- `_hwtimer_error_code_t HWTIMER_SYS_Stop (hwtimer_t *hwtimer)`  
*The timer stops counting after this function is called.*
- `uint32_t HWTIMER_SYS_GetModulo (hwtimer_t *hwtimer)`  
*The function returns period of the timer in sub-ticks.*
- `_hwtimer_error_code_t HWTIMER_SYS_GetTime (hwtimer_t *hwtimer, hwtimer_time_t *time)`  
*The function reads the current value of the hwtimer.*
- `uint32_t HWTIMER_SYS_GetTicks (hwtimer_t *hwtimer)`  
*The function reads the current value of the hwtimer.*
- `_hwtimer_error_code_t HWTIMER_SYS_RegisterCallback (hwtimer_t *hwtimer, hwtimer_callback_t callbackFunc, void *callbackData)`  
*Registers function to be called when the timer expires.*
- `_hwtimer_error_code_t HWTIMER_SYS_BlockCallback (hwtimer_t *hwtimer)`  
*The function is used to block callbacks in circumstances when execution of the callback function is undesired.*
- `_hwtimer_error_code_t HWTIMER_SYS_UnblockCallback (hwtimer_t *hwtimer)`  
*The function is used to unblock previously blocked callbacks.*
- `_hwtimer_error_code_t HWTIMER_SYS_CancelCallback (hwtimer_t *hwtimer)`  
*The function cancels pending callback, if any.*

## 41.1 Data Structure Documentation

### 41.1.1 struct hwtimer\_t

This structure defines a hwtimer. The context structure is passed to all API functions (besides other parameters).

#### Warning

Application should not access members of this structure directly.

#### See Also

- [HWTIMER\\_SYS\\_init](#)
- [HWTIMER\\_SYS\\_deinit](#)
- [HWTIMER\\_SYS\\_start](#)
- [HWTIMER\\_SYS\\_stop](#)
- [HWTIMER\\_SYS\\_set\\_freq](#)
- [HWTIMER\\_SYS\\_get\\_freq](#)
- [HWTIMER\\_SYS\\_set\\_period](#)
- [HWTIMER\\_SYS\\_get\\_period](#)
- [HWTIMER\\_SYS\\_get\\_modulo](#)
- [HWTIMER\\_SYS\\_get\\_time](#)
- [HWTIMER\\_SYS\\_get\\_ticks](#)
- [HWTIMER\\_SYS\\_callback\\_reg](#)
- [HWTIMER\\_SYS\\_callback\\_block](#)
- [HWTIMER\\_SYS\\_callback\\_unblock](#)
- [HWTIMER\\_SYS\\_callback\\_cancel](#)

## Data Fields

- `struct Hwtimer_devif * devif`  
*Pointer to device interface structure.*
- `clock_names_t clockName`  
*Clock identifier used for obtaining timer's source clock.*
- `uint32_t divider`  
*Actual total divider.*
- `uint32_t modulo`  
*Determine how many sub ticks are in one tick.*
- `volatile uint64_t ticks`  
*Number of elapsed ticks.*
- `hwtimer_callback_t callbackFunc`  
*Function pointer to be called when the timer expires.*
- `void * callbackData`  
*Arbitrary pointer passed as parameter to the callback function.*
- `volatile int callbackPending`

## Data Structure Documentation

- **int callbackBlocked**  
*Indicate pending callback.*
- **uint32\_t llContext [HWTIMER\_LL\_CONTEXT\_LEN]**  
*Indicate blocked callback.*  
*Private storage locations for arbitrary data keeping the context of the lower layer driver.*

### 41.1.1.0.0.58 Field Documentation

**41.1.1.0.0.58.1 clock\_names\_t hwtimer\_ptr\_t::clockName**

**41.1.1.0.0.58.2 hwtimer\_callback\_t hwtimer\_ptr\_t::callbackFunc**

**41.1.1.0.0.58.3 void\* hwtimer\_ptr\_t::callbackData**

**41.1.1.0.0.58.4 volatile int hwtimer\_ptr\_t::callbackPending**

If the timer overflows when callbacks are blocked the callback becomes pending.

**41.1.1.0.0.58.5 int hwtimer\_ptr\_t::callbackBlocked**

**41.1.1.0.0.58.6 uint32\_t hwtimer\_ptr\_t::llContext[HWTIMER\_LL\_CONTEXT\_LEN]**

### 41.1.2 struct hwtimer\_time\_t

Hwtimer time structure represents a time stamp consisting of timer elapsed periods (TICKS) and current value of the timer counter (subTicks).

See Also

[HWTIMER\\_SYS\\_GetTime](#)

### Data Fields

- **uint64\_t ticks**  
*Ticks of timer.*
- **uint32\_t subTicks**  
*Sub ticks of timer.*

### 41.1.3 struct hwtimer\_devif\_t

Each low layer driver exports instance of this structure initialized with pointers to API functions the driver implements. The functions themselves should be declared as static (not exported directly).

See Also

[HWTIMER\\_SYS\\_Init](#)  
[HWTIMER\\_SYS\\_Deinit](#)

## Data Fields

- [`hwtimer\_devif\_init\_t init`](#)  
*Function pointer to lower layer initialization.*
- [`hwtimer\_devif\_deinit\_t deinit`](#)  
*Function pointer to lower layer de-initialization.*
- [`hwtimer\_devif\_set\_div\_t setDiv`](#)  
*Function pointer to lower layer set divider functionality.*
- [`hwtimer\_devif\_start\_t start`](#)  
*Function pointer to lower layer start functionality.*
- [`hwtimer\_devif\_stop\_t stop`](#)  
*Function pointer to lower layer stop functionality.*
- [`hwtimer\_devif\_get\_time\_t getTime`](#)  
*Function pointer to lower layer get time functionality.*

## 41.2 Enumeration Type Documentation

### 41.2.1 `enum _hwtimer_error_code_t`

Enumerator

`kHwtimerSuccess` success  
`kHwtimerInvalidInput` invalid input parameter  
`kHwtimerInvalidPointer` invalid pointer  
`kHwtimerClockManagerError` clock manager return error  
`kHwtimerRegisterHandlerError` Interrupt handler registration returns error.  
`kHwtimerLockError` Error from synchronization object.  
`kHwtimerUnknown` unknown error

## 41.3 Function Documentation

### 41.3.1 `_hwtimer_error_code_t HWTIMER_SYS_Init( hwtimer_t * hwtimer, const hwtimer_devif_t * kDevif, uint32_t id, uint32_t isrPrior, void * data )`

The device interface pointer determines low layer driver to be used. Device interface structure is exported by each low layer driver and is opaque to the applications. Please refer to chapter concerning low layer driver below for details. Meaning of the numerical identifier varies depending on low layer driver used. Typically, it identifies particular timer channel to initialize. The initialization function has to be called prior using any other API function.

Parameters

---

## Function Documentation

|                     |                                                   |
|---------------------|---------------------------------------------------|
| <i>hwTimer[out]</i> | Returns initialized hwTimer structure handle.     |
| <i>kDevIf[in]</i>   | Structure determines low layer driver to be used. |
| <i>id[in]</i>       | Numerical identifier of the timer.                |
| <i>isrPrior[in]</i> | Priority of interrupt.                            |
| <i>data[in]</i>     | Specific data for low level of interrupt.         |

Returns

kHwTimerSuccess or an error code Returned from low level init function.  
kHwTimerInvalidInput When input parameter hwTimer is a NULL pointer  
kHwTimerInvalidPointer When device structure points to NULL.

Warning

The initialization function has to be called prior using any other API function.

See Also

[HWTIMER\\_SYS\\_deinit](#)  
[HWTIMER\\_SYS\\_start](#)  
[HWTIMER\\_SYS\\_stop](#)

### 41.3.2 `_hwTimer_error_code_t HWTIMER_SYS_Deinit( hwTimer_t * hwTimer )`

Calls lower layer stop function to stop timer, then calls low layer de-initialization function and afterwards invalidates hwTimer structure by clearing it.

Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwTimer[in]</i> | Pointer to hwTimer structure. |
|--------------------|-------------------------------|

Returns

kHwTimerSuccess or an error code Returned from low level DEINIT function.  
kHwTimerInvalidInput When input parameter hwTimer is a NULL pointer  
kHwTimerInvalidPointer When device structure points to NULL.

See Also

[HWTIMER\\_SYS\\_Init](#)  
[HWTIMER\\_SYS\\_Start](#)  
[HWTIMER\\_SYS\\_Stop](#)

#### 41.3.3 `_hwtimer_error_code_t HWTIMER_SYS_SetFreq ( hwtimer_t * hwtimer, clock_names_t clockName, uint32_t freq )`

Actual accuracy depends on the timer module. The function gets the value of the base frequency of the timer via clock manager, calculates required divider ratio and calls low layer driver to set up the timer accordingly. Call to this function might be expensive as it may require complex calculation to choose the best configuration of dividers. The actual complexity depends on timer module implementation. If there is only single divider or counter preload value (typical case) then there is no significant overhead.

##### Parameters

|                            |                                                           |
|----------------------------|-----------------------------------------------------------|
| <code>hwtimer[in]</code>   | Pointer to hwtimer structure.                             |
| <code>clockName[in]</code> | Clock identifier used for obtaining timer's source clock. |
| <code>freq[in]</code>      | Required frequency of timer in Hz.                        |

##### Returns

`kHwtimerSuccess` or an error code Returned from low level SETDIV function.  
`kHwtimerInvalidInput` When input parameter `hwtimer` is a NULL pointer  
`kHwtimerInvalidPointer` When device structure points to NULL.  
`kHwtimerClockManagerError` When Clock manager returns error.

##### See Also

[HWTIMER\\_SYS\\_GetFreq](#)  
[HWTIMER\\_SYS\\_SetPeriod](#)  
[HWTIMER\\_SYS\\_GetPeriod](#)  
[HWTIMER\\_SYS\\_GetModulo](#)  
[HWTIMER\\_SYS\\_GetTime](#)  
[HWTIMER\\_SYS\\_GetTicks](#)

#### 41.3.4 `_hwtimer_error_code_t HWTIMER_SYS_SetPeriod ( hwtimer_t * hwtimer, clock_names_t clockName, uint32_t period )`

The function provides an alternate way to set up the timer to desired period specified in microseconds rather than to frequency in Hz. The function gets the value of the base frequency of the timer via clock manager, calculates required divider ratio and calls low layer driver to set up the timer accordingly. Call to this function might be expensive as it may require complex calculation to choose the best configuration of dividers. The actual complexity depends on timer module implementation. If there is only single divider or counter preload value (typical case) then there is no significant overhead.

## Function Documentation

Parameters

|                      |                                                           |
|----------------------|-----------------------------------------------------------|
| <i>hwtimer[in]</i>   | Pointer to hwtimer structure.                             |
| <i>clockName[in]</i> | Clock identifier used for obtaining timer's source clock. |
| <i>period[in]</i>    | Required period of timer in micro seconds.                |

Returns

kHwtimerSuccess or an error code Returned from low level SETDIV function.

kHwtimerInvalidInput When input parameter hwtimer or his device structure are NULL pointers.

kHwtimerInvalidPointer When low level SETDIV function point to NULL.

kHwtimerClockManagerError When Clock manager returns error.

See Also

[HWTIMER\\_SYS\\_SetFreq](#)  
[HWTIMER\\_SYS\\_GetFreq](#)  
[HWTIMER\\_SYS\\_GetPeriod](#)  
[HWTIMER\\_SYS\\_GetModulo](#)  
[HWTIMER\\_SYS\\_GetTime](#)  
[HWTIMER\\_SYS\\_GetTicks](#)

### 41.3.5 **uint32\_t HWTIMER\_SYS\_GetFreq ( hwtimer\_t \* *hwtimer* )**

The function returns current frequency of the timer calculated from the base frequency and actual divider settings of the timer or zero in case of an error.

Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

Returns

frequency Already set frequency of hwtimer

0 When input parameter hwtimer is NULL pointer or his divider member is zero or clock manager returns error.

See Also

[HWTIMER\\_SYS\\_SetFreq](#)  
[HWTIMER\\_SYS\\_SetPeriod](#)  
[HWTIMER\\_SYS\\_GetPeriod](#)  
[HWTIMER\\_SYS\\_GetModulo](#)  
[HWTIMER\\_SYS\\_GetTime](#)  
[HWTIMER\\_SYS\\_GetTicks](#)

**41.3.6 uint32\_t HWTIMER\_SYS\_GetPeriod ( hwtimer\_t \* *hwtimer* )**

The function returns current period of the timer in microseconds calculated from the base frequency and actual divider settings of the timer.

## Function Documentation

Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

Returns

period Already set period of hwtimer.

0 Input parameter hwtimer is NULL pointer or clock manager returns error.

See Also

[HWTIMER\\_SYS\\_SetFreq](#)  
[HWTIMER\\_SYS\\_GetFreq](#)  
[HWTIMER\\_SYS\\_SetPeriod](#)  
[HWTIMER\\_SYS\\_GetModulo](#)  
[HWTIMER\\_SYS\\_GetTime](#)  
[HWTIMER\\_SYS\\_GetTicks](#)

### 41.3.7 `_hwtimer_error_code_t HWTIMER_SYS_Start ( hwtimer_t * hwtimer )`

The timer starts counting a new period generating interrupts every time the timer rolls over.

Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

Returns

kHwtimerSuccess or an error code Returned from low level START function.

kHwtimerInvalidInput When input parameter hwtimer is a NULL pointer

kHwtimerInvalidPointer When device structure points to NULL.

See Also

[HWTIMER\\_SYS\\_Init](#)  
[HWTIMER\\_SYS\\_Deinit](#)  
[HWTIMER\\_SYS\\_Stop](#)

### 41.3.8 `_hwtimer_error_code_t HWTIMER_SYS_Stop ( hwtimer_t * hwtimer )`

Pending interrupts and callbacks are cancelled.

Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

Returns

kHwtimerSuccess or an error code Returned from low level STOP function.

kHwtimerInvalidInput When input parameter hwtimer is a NULL pointer

kHwtimerInvalidPointer When device structure points to NULL.

See Also

[HWTIMER\\_SYS\\_Init](#)

[HWTIMER\\_SYS\\_Deinit](#)

[HWTIMER\\_SYS\\_Start](#)

#### 41.3.9 `uint32_t HWTIMER_SYS_GetModulo ( hwtimer_t * hwtimer )`

It is typically called after [HWTIMER\\_SYS\\_SetFreq\(\)](#) or [HWTIMER\\_SYS\\_SetPeriod\(\)](#) to obtain actual resolution of the timer in the current configuration.

Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

Returns

modulo resolution of hwtimer.

0 Input parameter hwtimer is NULL pointer.

See Also

[HWTIMER\\_SYS\\_SetFreq](#)

[HWTIMER\\_SYS\\_GetFreq](#)

[HWTIMER\\_SYS\\_SetPeriod](#)

[HWTIMER\\_SYS\\_GetPeriod](#)

[HWTIMER\\_SYS\\_GetTime](#)

[HWTIMER\\_SYS\\_GetTicks](#)

## Function Documentation

### 41.3.10 `_hwtimer_error_code_t HWTIMER_SYS_GetTime ( hwtimer_t * hwtimer, hwtimer_time_t * time )`

Elapsed periods(ticks) and current value of the timer counter (sub-ticks) are filled into the Hwtimer\_time structure. The sub-ticks number always counts up and is reset to zero when the timer overflows regardless of the counting direction of the underlying device.

## Parameters

|                    |                                                                                  |
|--------------------|----------------------------------------------------------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure.                                                    |
| <i>time[out]</i>   | Pointer to time structure. This value is filled with current value of the timer. |

## Returns

kHwtimerSuccess or an error code Returned from low level GET\_TIME function.

kHwtimerInvalidInput When input parameter hwtimer or input parameter time are NULL pointers.

kHwtimerInvalidPointer When device structure points to NULL.

## See Also

[HWTIMER\\_SYS\\_SetFreq](#)  
[HWTIMER\\_SYS\\_GetFreq](#)  
[HWTIMER\\_SYS\\_SetPeriod](#)  
[HWTIMER\\_SYS\\_GetPeriod](#)  
[HWTIMER\\_SYS\\_GetModulo](#)  
[HWTIMER\\_SYS\\_GetTicks](#)

**41.3.11 uint32\_t HWTIMER\_SYS\_GetTicks ( *hwtimer\_t \* hwtimer* )**

The returned value corresponds with lower 32 bits of elapsed periods (ticks). The value is guaranteed to be obtained atomically without necessity to mask timer interrupt. Lower layer driver is not involved at all, thus call to this function is considerably faster than HWTIMER\_SYS\_GetTime.

## Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

## Returns

Low 32 bits of 64 bit tick value.

0 When input parameter hwtimer is NULL pointer.

## See Also

[HWTIMER\\_SYS\\_SetFreq](#)  
[HWTIMER\\_SYS\\_GetFreq](#)  
[HWTIMER\\_SYS\\_SetPeriod](#)  
[HWTIMER\\_SYS\\_GetPeriod](#)  
[HWTIMER\\_SYS\\_GetModulo](#)  
[HWTIMER\\_SYS\\_GetTime](#)

## Function Documentation

**41.3.12 `_hwtimer_error_code_t HWTIMER_SYS_RegisterCallback ( hwtimer_t *  
hwtimer, hwtimer_callback_t callbackFunc, void * callbackData )`**

The callback\_data is arbitrary pointer passed as parameter to the callback function.

## Parameters

|                      |                                                            |
|----------------------|------------------------------------------------------------|
| <i>hwtimer[in]</i>   | Pointer to hwtimer structure.                              |
| <i>callback_func</i> | [in] Function pointer to be called when the timer expires. |
| <i>callback_data</i> | [in] Data pointer for the function callback_func.          |

## Returns

kHwtimerInvalidInput When input parameter hwtimer is NULL pointer.  
kHwtimerSuccess When registration callback succeed.

## Warning

This function must not be called from a callback routine.

## See Also

[HWTIMER\\_SYS\\_BlockCallback](#)  
[HWTIMER\\_SYS\\_UnblockCallback](#)  
[HWTIMER\\_SYS\\_CancelCallback](#)

#### 41.3.13 `_hwtimer_error_code_t HWTIMER_SYS_BlockCallback ( hwtimer_t * hwtimer )`

If the timer overflows when callbacks are blocked the callback becomes pending.

## Parameters

|                    |                               |
|--------------------|-------------------------------|
| <i>hwtimer[in]</i> | Pointer to hwtimer structure. |
|--------------------|-------------------------------|

## Returns

kHwtimerInvalidInput When input parameter hwtimer is NULL pointer.  
kHwtimerSuccess When callback block succeed.

## Warning

This function must not be called from a callback routine.

## See Also

[HWTIMER\\_SYS\\_RegCallback](#)  
[HWTIMER\\_SYS\\_UnblockCallback](#)  
[HWTIMER\\_SYS\\_CancelCallback](#)

## Function Documentation

### 41.3.14 `_hwtimer_error_code_t HWTIMER_SYS_UnblockCallback ( hwtimer_t * hwtimer )`

If there is a callback pending, it gets immediately executed. This function must not be called from a callback routine (it does not make sense to do so anyway as callback function never gets executed while callbacks are blocked).

#### Parameters

|                          |                               |
|--------------------------|-------------------------------|
| <code>hwtimer[in]</code> | Pointer to hwtimer structure. |
|--------------------------|-------------------------------|

#### Returns

`kHwtimerInvalidInput` When input parameter hwtimer is NULL pointer.  
`kHwtimerSuccess` When callback unblock succeed.

#### Warning

This function must not be called from a callback routine.

#### See Also

[HWTIMER\\_SYS\\_RegCallback](#)  
[HWTIMER\\_SYS\\_BlockCallback](#)  
[HWTIMER\\_SYS\\_CancelCallback](#)

### 41.3.15 `_hwtimer_error_code_t HWTIMER_SYS_CancelCallback ( hwtimer_t * hwtimer )`

#### Parameters

|                          |                               |
|--------------------------|-------------------------------|
| <code>hwtimer[in]</code> | Pointer to hwtimer structure. |
|--------------------------|-------------------------------|

#### Returns

`kHwtimerInvalidInput` When input parameter hwtimer is NULL pointer.  
`kHwtimerSuccess` When callback cancel succeed.

#### Warning

This function must not be called from a callback routine (it does not make sense to do so anyway as callback function never gets executed while callbacks are blocked).

See Also

[HWTIMER\\_SYS\\_RegCallback](#)  
[HWTIMER\\_SYS\\_BlockCallback](#)  
[HWTIMER\\_SYS\\_UnblockCallback](#)

## Chapter 42 Revision History

This table summarizes revisions to this document.

| Revision History |        |                     |
|------------------|--------|---------------------|
| Revision number  | Date   | Substantial changes |
| 1.0.0            | 7/2014 | Initial release     |

**How to Reach Us:**

**Home Page:**  
[freescale.com](http://freescale.com)

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

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

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

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

© 2014 Freescale Semiconductor, Inc.