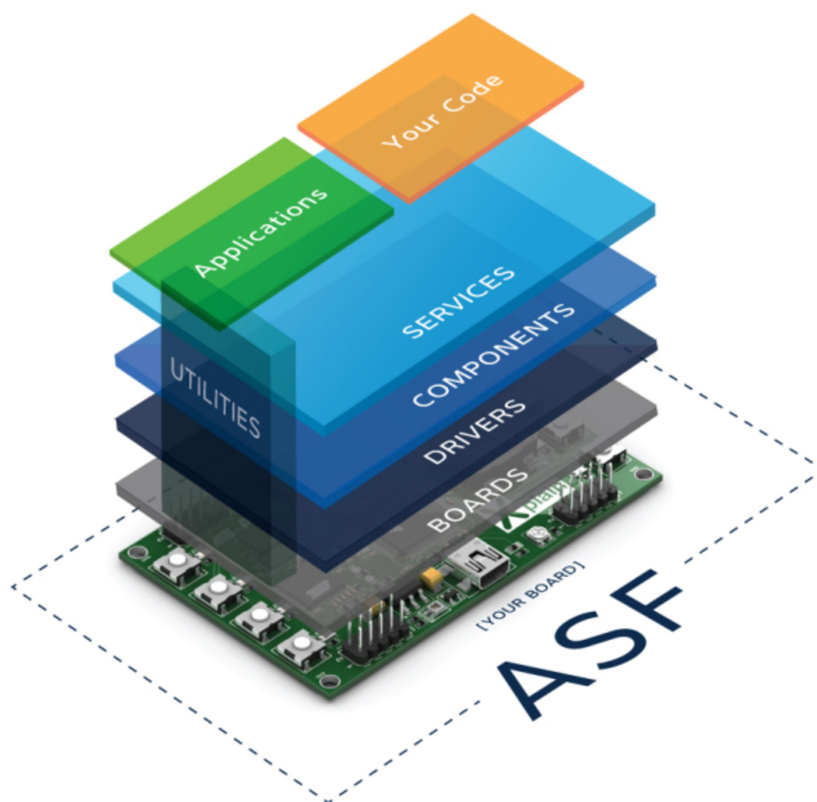


Preface

The Atmel® Software Framework (ASF) is a collection of free embedded software for Atmel microcontroller devices. It simplifies the usage of Atmel products, providing an abstraction to the hardware and high-value middleware. ASF is designed to be used for evaluation, prototyping, design, and production phases. ASF is integrated in the Atmel Studio IDE with a graphical user interface or available as a standalone package for several commercial and open source compilers. This document describes the API interfaces to the low level ASF module drivers of the device.



For more information on ASF, refer to the online documentation at <http://www.atmel.com/asf>.

Table of Contents

Preface.....	1
1. Software License.....	18
2. SAM Analog Comparator (AC) Driver.....	19
2.1. Prerequisites.....	19
2.2. Module Overview.....	19
2.2.1. Driver Feature Macro Definition.....	20
2.2.2. Window Comparators and Comparator Pairs.....	20
2.2.3. Positive and Negative Input MUXes.....	20
2.2.4. Output Filtering.....	20
2.2.5. Input Hysteresis.....	20
2.2.6. Single Shot and Continuous Sampling Modes.....	21
2.2.7. Events.....	21
2.2.8. Physical Connection.....	21
2.3. Special Considerations.....	22
2.4. Extra Information.....	22
2.5. Examples.....	22
2.6. API Overview.....	23
2.6.1. Variable and Type Definitions.....	23
2.6.2. Structure Definitions.....	23
2.6.3. Macro Definitions.....	25
2.6.4. Function Definitions.....	26
2.6.5. Enumeration Definitions.....	35
2.7. Extra Information for AC Driver.....	39
2.7.1. Acronyms.....	39
2.7.2. Dependencies.....	39
2.7.3. Errata.....	39
2.7.4. Module History.....	39
2.8. Examples for AC Driver.....	39
2.8.1. Quick Start Guide for AC - Basic.....	39
2.8.2. Quick Start Guide for AC - Callback.....	43
3. SAM Analog-to-Digital Converter (ADC) Driver.....	48
3.1. Prerequisites.....	48
3.2. Module Overview.....	48
3.2.1. Sample Clock Prescaler.....	49
3.2.2. ADC Resolution.....	49
3.2.3. Conversion Modes.....	49
3.2.4. Differential and Single-ended Conversion.....	49
3.2.5. Sample Time.....	50
3.2.6. Averaging.....	50
3.2.7. Offset and Gain Correction.....	51
3.2.8. Pin Scan.....	51
3.2.9. Window Monitor.....	51

3.2.10.	Events.....	51
3.3.	Special Considerations.....	52
3.4.	Extra Information.....	52
3.5.	Examples.....	52
3.6.	API Overview.....	52
3.6.1.	Variable and Type Definitions.....	52
3.6.2.	Structure Definitions.....	53
3.6.3.	Macro Definitions.....	55
3.6.4.	Function Definitions.....	56
3.6.5.	Enumeration Definitions.....	67
3.7.	Extra Information for ADC Driver.....	72
3.7.1.	Acronyms.....	72
3.7.2.	Dependencies.....	72
3.7.3.	Errata.....	73
3.7.4.	Module History.....	73
3.8.	Examples for ADC Driver.....	73
3.8.1.	Quick Start Guide for ADC - Basic.....	73
3.8.2.	Quick Start Guide for ADC - Callback.....	75
3.8.3.	Quick Start Guide for Using DMA with ADC/DAC.....	79
4.	SAM Advanced Encryption Standard (AES) Driver.....	86
4.1.	Prerequisites.....	86
4.2.	Module Overview.....	86
4.2.1.	Encryption and Decryption.....	87
4.2.2.	Hardware Countermeasures.....	87
4.2.3.	Galois Counter Mode (GCM).....	87
4.3.	Special Considerations.....	87
4.4.	Extra Information.....	87
4.5.	Examples.....	88
4.6.	API Overview.....	88
4.6.1.	Variable and Type Definitions.....	88
4.6.2.	Structure Definitions.....	88
4.6.3.	Macro Definitions.....	88
4.6.4.	Function Definitions.....	89
4.6.5.	Enumeration Definitions.....	97
4.7.	Extra Information for Advanced Encryption Standard.....	99
4.7.1.	Acronyms.....	99
4.7.2.	Dependencies.....	100
4.7.3.	Errata.....	100
4.7.4.	Module History.....	100
4.8.	Examples for Advanced Encryption Standard.....	100
4.8.1.	Quick Start Guide for AES - Basic.....	100
4.8.2.	Quick Start Guide for AES - Callback.....	105
4.8.3.	Quick Start Guide for AES - DMA.....	117
5.	SAM Brown Out Detector (BOD) Driver.....	123
5.1.	Prerequisites.....	123
5.2.	Module Overview.....	123
5.3.	Special Considerations.....	123

5.4.	Extra Information.....	123
5.5.	Examples.....	124
5.6.	API Overview.....	124
5.6.1.	Structure Definitions.....	124
5.6.2.	Function Definitions.....	125
5.6.3.	Enumeration Definitions.....	129
5.7.	Extra Information for BOD Driver.....	132
5.7.1.	Acronyms.....	132
5.7.2.	Dependencies.....	132
5.7.3.	Errata.....	132
5.7.4.	Module History.....	132
5.8.	Examples for BOD Driver.....	132
5.8.1.	Quick Start Guide for BOD - Basic.....	133
5.8.2.	Application Use Case for BOD - Application.....	134
6.	SAM Configurable Custom Logic (CCL) Driver.....	135
6.1.	Prerequisites.....	135
6.2.	Module Overview.....	135
6.3.	Special Considerations.....	135
6.4.	Extra Information.....	135
6.5.	Examples.....	136
6.6.	API Overview.....	136
6.6.1.	Structure Definitions.....	136
6.6.2.	Function Definitions.....	137
6.6.3.	Enumeration Definitions.....	139
6.7.	Extra Information for CCL Driver.....	141
6.7.1.	Acronyms.....	141
6.7.2.	Dependencies.....	141
6.7.3.	Errata.....	141
6.7.4.	Module History.....	141
6.8.	Examples for CCL Driver.....	141
6.8.1.	Quick Start Guide for CCL - Basic.....	142
7.	SAM Direct Memory Access Controller (DMAC) Driver.....	146
7.1.	Prerequisites.....	146
7.2.	Module Overview.....	146
7.2.1.	Driver Feature Macro Definition.....	147
7.2.2.	Terminology Used in DMAC Transfers.....	148
7.2.3.	DMA Channels.....	148
7.2.4.	DMA Triggers.....	148
7.2.5.	DMA Transfer Descriptor.....	148
7.2.6.	DMA Interrupts/Events.....	149
7.3.	Special Considerations.....	149
7.4.	Extra Information.....	149
7.5.	Examples.....	149
7.6.	API Overview.....	149
7.6.1.	Variable and Type Definitions.....	149
7.6.2.	Structure Definitions.....	150
7.6.3.	Macro Definitions.....	152

7.6.4.	Function Definitions.....	152
7.6.5.	Enumeration Definitions.....	159
7.7.	Extra Information for DMAC Driver.....	162
7.7.1.	Acronyms.....	162
7.7.2.	Dependencies.....	162
7.7.3.	Errata.....	162
7.7.4.	Module History.....	162
7.8.	Examples for DMAC Driver.....	162
7.8.1.	Quick Start Guide for Memory to Memory Data Transfer Using DMAC.....	163
8.	SAM Event System (EVENTS) Driver.....	167
8.1.	Prerequisites.....	167
8.2.	Module Overview.....	167
8.2.1.	Event Channels.....	168
8.2.2.	Event Users.....	168
8.2.3.	Edge Detection.....	168
8.2.4.	Path Selection.....	168
8.2.5.	Physical Connection.....	169
8.2.6.	Configuring Events.....	169
8.3.	Special Considerations.....	170
8.4.	Extra Information.....	170
8.5.	Examples.....	170
8.6.	API Overview.....	170
8.6.1.	Variable and Type Definitions.....	170
8.6.2.	Structure Definitions.....	170
8.6.3.	Macro Definitions.....	171
8.6.4.	Function Definitions.....	171
8.6.5.	Enumeration Definitions.....	180
8.7.	Extra Information for EVENTS Driver.....	181
8.7.1.	Acronyms.....	181
8.7.2.	Dependencies.....	181
8.7.3.	Errata.....	181
8.7.4.	Module History.....	181
8.8.	Examples for EVENTS Driver.....	181
8.8.1.	Quick Start Guide for EVENTS - Basic.....	181
8.8.2.	Quick Start Guide for EVENTS - Interrupt Hooks.....	184
9.	SAM External Interrupt (EXTINT) Driver.....	190
9.1.	Prerequisites.....	190
9.2.	Module Overview.....	190
9.2.1.	Logical Channels.....	190
9.2.2.	NMI Channels.....	191
9.2.3.	Input Filtering and Detection.....	191
9.2.4.	Events and Interrupts.....	191
9.2.5.	Physical Connection.....	191
9.3.	Special Considerations.....	192
9.4.	Extra Information.....	192
9.5.	Examples.....	192
9.6.	API Overview.....	192

9.6.1.	Variable and Type Definitions.....	192
9.6.2.	Structure Definitions.....	193
9.6.3.	Macro Definitions.....	194
9.6.4.	Function Definitions.....	194
9.6.5.	Enumeration Definitions.....	200
9.7.	Extra Information for EXTINT Driver.....	201
9.7.1.	Acronyms.....	201
9.7.2.	Dependencies.....	202
9.7.3.	Errata.....	202
9.7.4.	Module History.....	202
9.8.	Examples for EXTINT Driver.....	202
9.8.1.	Quick Start Guide for EXTINT - Basic.....	202
9.8.2.	Quick Start Guide for EXTINT - Callback.....	204
10.	SAM Frequency Meter (FREQM) Driver.....	207
10.1.	Prerequisites.....	207
10.2.	Module Overview.....	207
10.3.	Special Considerations.....	207
10.4.	Extra Information.....	208
10.5.	Examples.....	208
10.6.	API Overview.....	208
10.6.1.	Variable and Type Definitions.....	208
10.6.2.	Structure Definitions.....	208
10.6.3.	Function Definitions.....	209
10.6.4.	Enumeration Definitions.....	213
10.7.	Extra Information for FREQM Driver.....	214
10.7.1.	Acronyms.....	214
10.7.2.	Dependencies.....	214
10.7.3.	Errata.....	214
10.7.4.	Module History.....	214
10.8.	Examples for FREQM Driver.....	214
10.8.1.	Quick Start Guide for FREQM - Basic.....	215
10.8.2.	Quick Start Guide for FREQM - Callback.....	217
11.	SAM Non-Volatile Memory (NVM) Driver.....	221
11.1.	Prerequisites.....	221
11.2.	Module Overview.....	221
11.2.1.	Driver Feature Macro Definition.....	221
11.2.2.	Memory Regions.....	221
11.2.3.	Region Lock Bits.....	223
11.2.4.	Read/Write.....	223
11.3.	Special Considerations.....	223
11.3.1.	Page Erasure.....	223
11.3.2.	Clocks.....	223
11.3.3.	Security Bit.....	223
11.4.	Extra Information.....	223
11.5.	Examples.....	224
11.6.	API Overview.....	224
11.6.1.	Structure Definitions.....	224

11.6.2.	Macro Definitions.....	226
11.6.3.	Function Definitions.....	226
11.6.4.	Enumeration Definitions.....	233
11.7.	Extra Information for NVM Driver.....	238
11.7.1.	Acronyms.....	238
11.7.2.	Dependencies.....	238
11.7.3.	Errata.....	238
11.7.4.	Module History.....	238
11.8.	Examples for NVM Driver.....	239
11.8.1.	Quick Start Guide for NVM - Basic.....	239
12.	SAM Peripheral Access Controller (PAC) Driver.....	242
12.1.	Prerequisites.....	242
12.2.	Module Overview.....	242
12.2.1.	Locking Scheme.....	242
12.2.2.	Recommended Implementation.....	243
12.2.3.	Why Disable Interrupts.....	245
12.2.4.	Run-away Code.....	246
12.2.5.	Faulty Module Pointer.....	248
12.2.6.	Use of <code>__no_inline</code>	248
12.2.7.	Physical Connection.....	248
12.3.	Special Considerations.....	248
12.3.1.	Non-Writable Registers.....	248
12.3.2.	Reading Lock State.....	248
12.4.	Extra Information.....	249
12.5.	Examples.....	249
12.6.	API Overview.....	249
12.6.1.	Macro Definitions.....	249
12.6.2.	Function Definitions.....	250
12.7.	Extra Information for PAC Driver.....	252
12.7.1.	Acronyms.....	252
12.7.2.	Dependencies.....	253
12.7.3.	Errata.....	253
12.7.4.	Module History.....	253
12.8.	Examples for PAC Driver.....	253
12.8.1.	Quick Start Guide for PAC - Basic.....	253
12.9.	List of Non-Write Protected Registers.....	255
13.	SAM Port (PORT) Driver.....	258
13.1.	Prerequisites.....	258
13.2.	Module Overview.....	258
13.2.1.	Driver Feature Macro Definition.....	258
13.2.2.	Physical and Logical GPIO Pins.....	258
13.2.3.	Physical Connection.....	259
13.3.	Special Considerations.....	259
13.4.	Extra Information.....	259
13.5.	Examples.....	259
13.6.	API Overview.....	260
13.6.1.	Structure Definitions.....	260

13.6.2.	Macro Definitions.....	260
13.6.3.	Function Definitions.....	261
13.6.4.	Enumeration Definitions.....	267
13.7.	Extra Information for PORT Driver.....	268
13.7.1.	Acronyms.....	268
13.7.2.	Dependencies.....	268
13.7.3.	Errata.....	268
13.7.4.	Module History.....	268
13.8.	Examples for PORT Driver.....	268
13.8.1.	Quick Start Guide for PORT - Basic.....	269
14.	SAM RTC Count (RTC COUNT) Driver.....	271
14.1.	Prerequisites.....	271
14.2.	Module Overview.....	271
14.2.1.	Driver Feature Macro Definition.....	272
14.3.	Compare and Overflow.....	272
14.3.1.	Periodic Events.....	272
14.3.2.	Digital Frequency Correction.....	273
14.3.3.	RTC Tamper Detect.....	273
14.4.	Special Considerations.....	274
14.4.1.	Clock Setup.....	274
14.5.	Extra Information.....	275
14.6.	Examples.....	275
14.7.	API Overview.....	275
14.7.1.	Structure Definitions.....	275
14.7.2.	Macro Definitions.....	276
14.7.3.	Function Definitions.....	278
14.7.4.	Enumeration Definitions.....	291
14.8.	RTC Tamper Detect.....	295
14.9.	Extra Information for RTC COUNT Driver.....	295
14.9.1.	Acronyms.....	295
14.9.2.	Dependencies.....	296
14.9.3.	Errata.....	296
14.9.4.	Module History.....	296
14.10.	Examples for RTC (COUNT) Driver.....	296
14.10.1.	Quick Start Guide for RTC (COUNT) - Basic.....	296
14.10.2.	Quick Start Guide for RTC (COUNT) - Callback.....	299
14.10.3.	Quick Start Guide for RTC Tamper with DMA.....	301
15.	SAM RTC Calendar (RTC CAL) Driver.....	306
15.1.	Prerequisites.....	306
15.2.	Module Overview.....	306
15.2.1.	Driver Feature Macro Definition.....	307
15.2.2.	Alarms and Overflow.....	307
15.2.3.	Periodic Events.....	307
15.2.4.	Digital Frequency Correction.....	308
15.2.5.	RTC Tamper Detect.....	308
15.3.	Special Considerations.....	308
15.3.1.	Year Limit.....	308

15.3.2.	Clock Setup.....	309
15.4.	Extra Information.....	310
15.5.	Examples.....	310
15.6.	API Overview.....	310
15.6.1.	Structure Definitions.....	310
15.6.2.	Macro Definitions.....	312
15.6.3.	Function Definitions.....	313
15.6.4.	Enumeration Definitions.....	326
15.7.	RTC Tamper Detect.....	330
15.8.	Extra Information for RTC (CAL) Driver.....	331
15.8.1.	Acronyms.....	331
15.8.2.	Dependencies.....	331
15.8.3.	Errata.....	331
15.8.4.	Module History.....	331
15.9.	Examples for RTC CAL Driver.....	331
15.9.1.	Quick Start Guide for RTC (CAL) - Basic.....	332
15.9.2.	Quick Start Guide for RTC (CAL) - Callback.....	334
16.	SAM Serial USART (SERCOM USART) Driver.....	339
16.1.	Prerequisites.....	339
16.2.	Module Overview.....	339
16.2.1.	Driver Feature Macro Definition.....	340
16.2.2.	Frame Format.....	340
16.2.3.	Synchronous Mode.....	341
16.2.4.	Asynchronous Mode.....	341
16.2.5.	Parity.....	341
16.2.6.	GPIO Configuration.....	342
16.3.	Special Considerations.....	342
16.4.	Extra Information.....	342
16.5.	Examples.....	342
16.6.	API Overview.....	342
16.6.1.	Variable and Type Definitions.....	342
16.6.2.	Structure Definitions.....	342
16.6.3.	Macro Definitions.....	346
16.6.4.	Function Definitions.....	347
16.6.5.	Enumeration Definitions.....	360
16.7.	Extra Information for SERCOM USART Driver.....	366
16.7.1.	Acronyms.....	366
16.7.2.	Dependencies.....	366
16.7.3.	Errata.....	367
16.7.4.	Module History.....	367
16.8.	Examples for SERCOM USART Driver.....	367
16.8.1.	Quick Start Guide for SERCOM USART - Basic.....	368
16.8.2.	Quick Start Guide for SERCOM USART - Callback.....	370
16.8.3.	Quick Start Guide for Using DMA with SERCOM USART.....	372
16.8.4.	Quick Start Guide for SERCOM USART LIN.....	378
16.9.	SERCOM USART MUX Settings.....	383
17.	SAM I ² C Master Mode (SERCOM I2C) Driver.....	384

17.1.	Prerequisites.....	384
17.2.	Module Overview.....	384
17.2.1.	Driver Feature Macro Definition.....	385
17.2.2.	Functional Description.....	385
17.2.3.	Bus Topology.....	385
17.2.4.	Transactions.....	386
17.2.5.	Multi Master.....	387
17.2.6.	Bus States.....	388
17.2.7.	Bus Timing.....	388
17.2.8.	Operation in Sleep Modes.....	389
17.3.	Special Considerations.....	389
17.3.1.	Interrupt-driven Operation.....	389
17.4.	Extra Information.....	389
17.5.	Examples.....	389
17.6.	API Overview.....	390
17.6.1.	Structure Definitions.....	390
17.6.2.	Macro Definitions.....	391
17.6.3.	Function Definitions.....	392
17.6.4.	Enumeration Definitions.....	406
17.7.	Extra Information for SERCOM I ² C Driver.....	408
17.7.1.	Acronyms.....	408
17.7.2.	Dependencies.....	408
17.7.3.	Errata.....	408
17.7.4.	Module History.....	408
17.8.	Examples for SERCOM I ² C Driver.....	409
17.8.1.	Quick Start Guide for SERCOM I ² C Master - Basic.....	409
17.8.2.	Quick Start Guide for SERCOM I ² C Master - Callback.....	412
17.8.3.	Quick Start Guide for Using DMA with SERCOM I ² C Master.....	415
18.	SAM I ² C Slave Mode (SERCOM I2C) Driver.....	420
18.1.	Prerequisites.....	420
18.2.	Module Overview.....	420
18.2.1.	Driver Feature Macro Definition.....	421
18.2.2.	Functional Description.....	421
18.2.3.	Bus Topology.....	421
18.2.4.	Transactions.....	422
18.2.5.	Multi Master.....	423
18.2.6.	Bus States.....	424
18.2.7.	Bus Timing.....	424
18.2.8.	Operation in Sleep Modes.....	425
18.3.	Special Considerations.....	425
18.3.1.	Interrupt-driven Operation.....	425
18.4.	Extra Information.....	425
18.5.	Examples.....	425
18.6.	API Overview.....	426
18.6.1.	Structure Definitions.....	426
18.6.2.	Macro Definitions.....	427
18.6.3.	Function Definitions.....	429
18.6.4.	Enumeration Definitions.....	440

18.7.	Extra Information for SERCOM I ² C Driver.....	442
18.7.1.	Acronyms.....	442
18.7.2.	Dependencies.....	442
18.7.3.	Errata.....	442
18.7.4.	Module History.....	442
18.8.	Examples for SERCOM I ² C Driver.....	442
18.8.1.	Quick Start Guide for SERCOM I ² C Slave - Basic.....	443
18.8.2.	Quick Start Guide for SERCOM I ² C Slave - Callback.....	445
18.8.3.	Quick Start Guide for Using DMA with SERCOM I ² C Slave.....	448
19.	SAM Serial Peripheral Interface (SERCOM SPI) Driver.....	453
19.1.	Prerequisites.....	453
19.2.	Module Overview.....	453
19.2.1.	Driver Feature Macro Definition.....	454
19.2.2.	SPI Bus Connection.....	454
19.2.3.	SPI Character Size.....	455
19.2.4.	Master Mode.....	455
19.2.5.	Slave Mode.....	455
19.2.6.	Data Modes.....	455
19.2.7.	SERCOM Pads.....	456
19.2.8.	Operation in Sleep Modes.....	456
19.2.9.	Clock Generation.....	456
19.3.	Special Considerations.....	456
19.3.1.	pinmux Settings.....	456
19.4.	Extra Information.....	457
19.5.	Examples.....	457
19.6.	API Overview.....	457
19.6.1.	Variable and Type Definitions.....	457
19.6.2.	Structure Definitions.....	457
19.6.3.	Macro Definitions.....	459
19.6.4.	Function Definitions.....	460
19.6.5.	Enumeration Definitions.....	475
19.7.	Extra Information for SERCOM SPI Driver.....	479
19.7.1.	Acronyms.....	479
19.7.2.	Dependencies.....	479
19.7.3.	Workarounds Implemented by Driver.....	479
19.7.4.	Module History.....	479
19.8.	Examples for SERCOM SPI Driver.....	480
19.8.1.	Quick Start Guide for SERCOM SPI Master - Polled.....	480
19.8.2.	Quick Start Guide for SERCOM SPI Slave - Polled.....	483
19.8.3.	Quick Start Guide for SERCOM SPI Master - Callback.....	486
19.8.4.	Quick Start Guide for SERCOM SPI Slave - Callback.....	490
19.8.5.	Quick Start Guide for Using DMA with SERCOM SPI.....	494
19.9.	MUX Settings.....	503
19.9.1.	Master Mode Settings.....	503
19.9.2.	Slave Mode Settings.....	504
20.	SAM Segment Liquid Crystal Display(SLCD) Controller.....	506
20.1.	Prerequisites.....	506

20.2.	Module Overview.....	506
20.2.1.	Display Overview.....	506
20.3.	Special Considerations.....	506
20.3.1.	I/O Lines.....	506
20.3.2.	Power Management.....	507
20.4.	Extra Information.....	507
20.5.	Examples.....	507
20.6.	API Overview.....	507
20.6.1.	Variable and Type Definitions.....	507
20.6.2.	Structure Definitions.....	507
20.6.3.	Macro Definitions.....	509
20.6.4.	Function Definitions.....	509
20.6.5.	Enumeration Definitions.....	524
20.7.	Extra Information for SLCD.....	526
20.7.1.	Acronyms.....	526
20.7.2.	Dependencies.....	526
20.7.3.	Errata.....	526
20.7.4.	Module History.....	526
20.8.	Examples for SLCD.....	526
20.8.1.	Quick Start Guide for SLCD.....	526
21.	SAM System (SYSTEM) Driver.....	535
21.1.	Prerequisites.....	535
21.2.	Module Overview.....	535
21.2.1.	Voltage Regulator.....	535
21.2.2.	Battery Backup Power Switch.....	535
21.2.3.	Voltage References.....	536
21.2.4.	System Reset Cause.....	536
21.2.5.	Performance Level.....	536
21.2.6.	Power Domain Gating.....	537
21.2.7.	RAMs Low Power Mode.....	538
21.2.8.	Sleep Modes.....	539
21.3.	Special Considerations.....	539
21.4.	Extra Information.....	539
21.5.	Examples.....	539
21.6.	API Overview.....	540
21.6.1.	Structure Definitions.....	540
21.6.2.	Function Definitions.....	541
21.6.3.	Enumeration Definitions.....	550
21.7.	Examples for SYSTEM Driver.....	553
21.7.1.	Quick Start Guide for Power Driver.....	553
21.8.	Extra Information for SYSTEM Driver.....	558
21.8.1.	Acronyms.....	558
21.8.2.	Dependencies.....	559
21.8.3.	Errata.....	559
21.8.4.	Module History.....	559
22.	SAM System Clock Management (SYSTEM CLOCK) Driver.....	560
22.1.	Prerequisites.....	560

22.2.	Module Overview.....	560
22.2.1.	Clock Sources.....	560
22.2.2.	CPU / Bus Clocks.....	561
22.2.3.	Clock Masking.....	561
22.2.4.	Generic Clocks.....	561
22.3.	Special Considerations.....	563
22.4.	Extra Information.....	563
22.5.	Examples.....	563
22.6.	API Overview.....	563
22.6.1.	Structure Definitions.....	563
22.6.2.	Function Definitions.....	567
22.6.3.	Enumeration Definitions.....	582
22.7.	Extra Information for SYSTEM CLOCK Driver.....	589
22.7.1.	Acronyms.....	589
22.7.2.	Dependencies.....	590
22.7.3.	Errata.....	590
22.7.4.	Module History.....	590
22.8.	Examples for System Clock Driver.....	590
22.8.1.	Quick Start Guide for SYSTEM CLOCK - Basic.....	590
22.8.2.	Quick Start Guide for SYSTEM CLOCK - GCLK Configuration.....	593
23.	SAM System Interrupt (SYSTEM INTERRUPT) Driver.....	596
23.1.	Prerequisites.....	596
23.2.	Module Overview.....	596
23.2.1.	Critical Sections.....	596
23.2.2.	Software Interrupts.....	596
23.3.	Special Considerations.....	597
23.4.	Extra Information.....	597
23.5.	Examples.....	597
23.6.	API Overview.....	597
23.6.1.	Function Definitions.....	597
23.6.2.	Enumeration Definitions.....	602
23.7.	Extra Information for SYSTEM INTERRUPT Driver.....	615
23.7.1.	Acronyms.....	615
23.7.2.	Dependencies.....	616
23.7.3.	Errata.....	616
23.7.4.	Module History.....	616
23.8.	Examples for SYSTEM INTERRUPT Driver.....	616
23.8.1.	Quick Start Guide for SYSTEM INTERRUPT - Critical Section Use Case.....	616
23.8.2.	Quick Start Guide for SYSTEM INTERRUPT - Enable Module Interrupt Use Case.....	617
24.	SAM System Pin Multiplexer (SYSTEM PINMUX) Driver.....	618
24.1.	Prerequisites.....	618
24.2.	Module Overview.....	618
24.2.1.	Driver Feature Macro Definition.....	618
24.2.2.	Physical and Logical GPIO Pins.....	618
24.2.3.	Peripheral Multiplexing.....	619
24.2.4.	Special Pad Characteristics.....	619
24.2.5.	Physical Connection.....	619

24.3.	Special Considerations.....	620
24.4.	Extra Information.....	620
24.5.	Examples.....	620
24.6.	API Overview.....	620
24.6.1.	Structure Definitions.....	620
24.6.2.	Macro Definitions.....	621
24.6.3.	Function Definitions.....	621
24.6.4.	Enumeration Definitions.....	624
24.7.	Extra Information for SYSTEM PINMUX Driver.....	626
24.7.1.	Acronyms.....	626
24.7.2.	Dependencies.....	626
24.7.3.	Errata.....	626
24.7.4.	Module History.....	626
24.8.	Examples for SYSTEM PINMUX Driver.....	626
24.8.1.	Quick Start Guide for SYSTEM PINMUX - Basic.....	626
25.	SAM Timer/Counter (TC) Driver.....	629
25.1.	Prerequisites.....	629
25.2.	Module Overview.....	629
25.2.1.	Driver Feature Macro Definition.....	631
25.2.2.	Functional Description.....	631
25.2.3.	Timer/Counter Size.....	631
25.2.4.	Clock Settings.....	632
25.2.5.	Compare Match Operations.....	633
25.2.6.	One-shot Mode.....	635
25.3.	Special Considerations.....	635
25.4.	Extra Information.....	636
25.5.	Examples.....	636
25.6.	API Overview.....	636
25.6.1.	Variable and Type Definitions.....	636
25.6.2.	Structure Definitions.....	636
25.6.3.	Macro Definitions.....	639
25.6.4.	Function Definitions.....	641
25.6.5.	Enumeration Definitions.....	650
25.7.	Extra Information for TC Driver.....	653
25.7.1.	Acronyms.....	653
25.7.2.	Dependencies.....	653
25.7.3.	Errata.....	653
25.7.4.	Module History.....	653
25.8.	Examples for TC Driver.....	654
25.8.1.	Quick Start Guide for TC - Basic.....	654
25.8.2.	Quick Start Guide for TC - Match Frequency Wave Generation.....	657
25.8.3.	Quick Start Guide for TC - Timer.....	659
25.8.4.	Quick Start Guide for TC - Callback.....	662
25.8.5.	Quick Start Guide for Using DMA with TC.....	666
26.	SAM Timer Counter for Control Applications (TCC) Driver.....	673
26.1.	Prerequisites.....	673
26.2.	Module Overview.....	673

26.2.1.	Functional Description.....	674
26.2.2.	Base Timer/Counter.....	675
26.2.3.	Capture Operations.....	677
26.2.4.	Compare Match Operation.....	678
26.2.5.	Waveform Extended Controls.....	679
26.2.6.	Double and Circular Buffering.....	681
26.2.7.	Sleep Mode.....	681
26.3.	Special Considerations.....	681
26.3.1.	Driver Feature Macro Definition.....	681
26.3.2.	Module Features.....	681
26.3.3.	Channels vs. Pinouts.....	682
26.4.	Extra Information.....	682
26.5.	Examples.....	682
26.6.	API Overview.....	682
26.6.1.	Variable and Type Definitions.....	682
26.6.2.	Structure Definitions.....	683
26.6.3.	Macro Definitions.....	688
26.6.4.	Function Definitions.....	690
26.6.5.	Enumeration Definitions.....	705
26.7.	Extra Information for TCC Driver.....	715
26.7.1.	Acronyms.....	715
26.7.2.	Dependencies.....	715
26.7.3.	Errata.....	715
26.7.4.	Module History.....	715
26.8.	Examples for TCC Driver.....	715
26.8.1.	Quick Start Guide for TCC - Basic.....	716
26.8.2.	Quick Start Guide for TCC - Double Buffering and Circular.....	719
26.8.3.	Quick Start Guide for TCC - Timer.....	722
26.8.4.	Quick Start Guide for TCC - Callback.....	725
26.8.5.	Quick Start Guide for TCC - Non-Recoverable Fault.....	728
26.8.6.	Quick Start Guide for TCC - Recoverable Fault.....	737
26.8.7.	Quick Start Guide for Using DMA with TCC.....	746
27.	SAM True Random Number Generator (TRNG) Driver.....	756
27.1.	Prerequisites.....	756
27.2.	Module Overview.....	756
27.3.	Special Considerations.....	756
27.4.	Extra Information.....	756
27.5.	Examples.....	757
27.6.	API Overview.....	757
27.6.1.	Variable and Type Definitions.....	757
27.6.2.	Structure Definitions.....	757
27.6.3.	Function Definitions.....	758
27.6.4.	Enumeration Definitions.....	763
27.7.	Extra Information for TRNG Driver.....	764
27.7.1.	Acronyms.....	764
27.7.2.	Dependencies.....	764
27.7.3.	Errata.....	764
27.7.4.	Module History.....	764

27.8.	Examples for TRNG Driver.....	764
27.8.1.	Quick Start Guide for TRNG - Basic.....	764
27.8.2.	Quick Start Guide for TRNG - Callback.....	766
28.	SAM Watchdog (WDT) Driver.....	769
28.1.	Prerequisites.....	769
28.2.	Module Overview.....	769
28.2.1.	Locked Mode.....	770
28.2.2.	Window Mode.....	770
28.2.3.	Early Warning.....	770
28.2.4.	Physical Connection.....	770
28.3.	Special Considerations.....	770
28.4.	Extra Information.....	771
28.5.	Examples.....	771
28.6.	API Overview.....	771
28.6.1.	Variable and Type Definitions.....	771
28.6.2.	Structure Definitions.....	771
28.6.3.	Function Definitions.....	772
28.6.4.	Enumeration Definitions.....	776
28.7.	Extra Information for WDT Driver.....	777
28.7.1.	Acronyms.....	777
28.7.2.	Dependencies.....	777
28.7.3.	Errata.....	777
28.7.4.	Module History.....	777
28.8.	Examples for WDT Driver.....	777
28.8.1.	Quick Start Guide for WDT - Basic.....	777
28.8.2.	Quick Start Guide for WDT - Callback.....	779
29.	SAM EEPROM Emulator (EEPROM) Service.....	782
29.1.	Prerequisites.....	782
29.2.	Module Overview.....	782
29.2.1.	Implementation Details.....	783
29.2.2.	Memory Layout.....	785
29.3.	Special Considerations.....	786
29.3.1.	NVM Controller Configuration.....	786
29.3.2.	Logical EEPROM Page Size.....	787
29.3.3.	Committing of the Write Cache.....	787
29.4.	Extra Information.....	787
29.5.	Examples.....	787
29.6.	API Overview.....	787
29.6.1.	Structure Definitions.....	787
29.6.2.	Macro Definitions.....	788
29.6.3.	Function Definitions.....	788
29.7.	Extra Information.....	792
29.7.1.	Acronyms.....	792
29.7.2.	Dependencies.....	792
29.7.3.	Errata.....	792
29.7.4.	Module History.....	792
29.8.	Examples for Emulated EEPROM Service.....	793

29.8.1. Quick Start Guide for the Emulated EEPROM Module - Basic Use Case.....	793
30. SAM Read While Write EEPROM (RWW EEPROM) Emulator Service.....	796
30.1. Prerequisites.....	796
30.2. Module Overview.....	796
30.2.1. Implementation Details.....	797
30.2.2. Memory Layout.....	799
30.3. Special Considerations.....	801
30.3.1. NVM Controller Configuration.....	801
30.3.2. Logical RWW EEPROM Page Size.....	801
30.3.3. Committing of the Write Cache.....	801
30.3.4. RWW EEPROM Page Checksum.....	801
30.4. Extra Information.....	801
30.5. Examples.....	802
30.6. API Overview.....	802
30.6.1. Structure Definitions.....	802
30.6.2. Macro Definitions.....	802
30.6.3. Function Definitions.....	803
30.6.4. Enumeration Definitions.....	807
30.7. Extra Information.....	807
30.7.1. Acronyms.....	807
30.7.2. Dependencies.....	807
30.7.3. Errata.....	807
30.7.4. Module History.....	807
30.8. Examples for Emulated RWW EEPROM Service.....	808
30.8.1. Quick Start Guide for the Emulated RWW EEPROM Module - Basic Use Case.....	808
31. Document Revision History.....	811

1. Software License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of Atmel may not be used to endorse or promote products derived from this software without specific prior written permission.
4. This software may only be redistributed and used in connection with an Atmel microcontroller product.

THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2. SAM Analog Comparator (AC) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Analog Comparator functionality, for the comparison of analog voltages against a known reference voltage to determine its relative level. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripherals are used by this module:

- AC (Analog Comparator)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

2.1. Prerequisites

There are no prerequisites for this module.

2.2. Module Overview

The Analog Comparator module provides an interface for the comparison of one or more analog voltage inputs (sourced from external or internal inputs) against a known reference voltage, to determine if the unknown voltage is higher or lower than the reference. Additionally, window functions are provided so that two comparators can be connected together to determine if an input is below, inside, above, or outside the two reference points of the window.

Each comparator requires two analog input voltages, a positive and negative channel input. The result of the comparison is a binary `true` if the comparator's positive channel input is higher than the comparator's negative input channel, and `false` if otherwise.

2.2.1. Driver Feature Macro Definition

Driver feature macro	Supported devices
FEATURE_AC_HYSTERSIS_LEVEL	SAM L21/L22/C20/C21
FEATURE_AC_SYNCBUSY_SCHEME_VERSION_2	SAM L21/L22/C20/C21
FEATURE_AC_RUN_IN_STANDY_EACH_COMPARATOR	SAM L21/L22/C20/C21
FEATURE_AC_RUN_IN_STANDY_PAIR_COMPARATOR	SAM D20/L22/D21/D10/D11/R21/DAX

Note: The specific features are only available in the driver when the selected device supports those features.

2.2.2. Window Comparators and Comparator Pairs

Each comparator module contains one or more comparator pairs, a set of two distinct comparators which can be used independently or linked together for Window Comparator mode. In this latter mode, the two comparator units in a comparator pair are linked together to allow the module to detect if an input voltage is below, inside, above, or outside a window set by the upper and lower threshold voltages set by the two comparators. If not required, window comparison mode can be turned off and the two comparator units can be configured and used separately.

2.2.3. Positive and Negative Input MUXes

Each comparator unit requires two input voltages, a positive and a negative channel (note that these names refer to the logical operation that the unit performs, and both voltages should be above GND), which are then compared with one another. Both the positive and the negative channel inputs are connected to a pair of multiplexers (MUXes), which allows one of several possible inputs to be selected for each comparator channel.

The exact channels available for each comparator differ for the positive and the negative inputs, but the same MUX choices are available for all comparator units (i.e. all positive MUXes are identical, all negative MUXes are identical). This allows the user application to select which voltages are compared to one another.

When used in window mode, both comparators in the window pair should have their positive channel input MUXes configured to the same input channel, with the negative channel input MUXes used to set the lower and upper window bounds.

2.2.4. Output Filtering

The output of each comparator unit can either be used directly with no filtering (giving a lower latency signal, with potentially more noise around the comparison threshold) or be passed through a multiple stage digital majority filter. Several filter lengths are available, with the longer stages producing a more stable result, at the expense of a higher latency.

When output filtering is used in single shot mode, a single trigger of the comparator will automatically perform the required number of samples to produce a correctly filtered result.

2.2.5. Input Hysteresis

To prevent unwanted noise around the threshold where the comparator unit's positive and negative input channels are close in voltage to one another, an optional hysteresis can be used to widen the point at

which the output result flips. This mode will prevent a change in the comparison output unless the inputs cross one another beyond the hysteresis gap introduced by this mode.

2.2.6. Single Shot and Continuous Sampling Modes

Comparators can be configured to run in either Single Shot or Continuous sampling modes; when in Single Shot mode, the comparator will only perform a comparison (and any resulting filtering, see [Output Filtering](#)) when triggered via a software or event trigger. This mode improves the power efficiency of the system by only performing comparisons when actually required by the application.

For systems requiring a lower latency or more frequent comparisons, continuous mode will place the comparator into continuous sampling mode, which increases the module's power consumption, but decreases the latency between each comparison result by automatically performing a comparison on every cycle of the module's clock.

2.2.7. Events

Each comparator unit is capable of being triggered by both software and hardware triggers. Hardware input events allow for other peripherals to automatically trigger a comparison on demand - for example, a timer output event could be used to trigger comparisons at a desired regular interval.

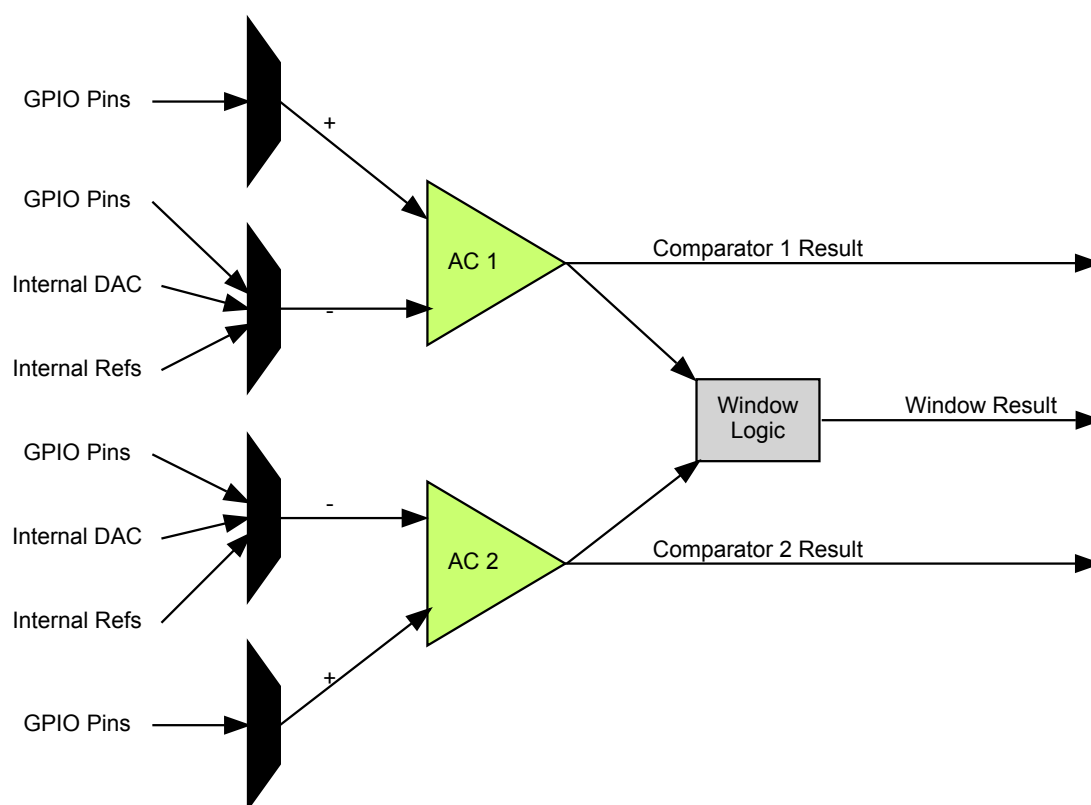
The module's output events can similarly be used to trigger other hardware modules each time a new comparison result is available. This scheme allows for reduced levels of CPU usage in an application and lowers the overall system response latency by directly triggering hardware peripherals from one another without requiring software intervention.

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the input event of another. For more information on event routing, refer to the event driver documentation.

2.2.8. Physical Connection

Physically, the modules are interconnected within the device as shown in [Figure 2-1](#).

Figure 2-1. Physical Connection



2.3. Special Considerations

The number of comparator pairs (and, thus, window comparators) within a single hardware instance of the Analog Comparator module is device-specific. Some devices will contain a single comparator pair, while others may have two pairs; refer to your device specific datasheet for details.

2.4. Extra Information

For extra information, see [Extra Information for AC Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

2.5. Examples

For a list of examples related to this driver, see [Examples for AC Driver](#).

2.6. API Overview

2.6.1. Variable and Type Definitions

2.6.1.1. Type `ac_callback_t`

```
typedef void(* ac_callback_t )(struct ac_module *const module_inst)
```

Type definition for a AC module callback function.

2.6.2. Structure Definitions

2.6.2.1. Struct `ac_chan_config`

Configuration structure for a comparator channel, to configure the input and output settings of the comparator.

Table 2-1. Members

Type	Name	Description
bool	enable_hysteresis	When <code>true</code> , hysteresis mode is enabled on the comparator inputs
enum <code>ac_chan_filter</code>	filter	Filtering mode for the comparator output, when the comparator is used in a supported mode
enum <code>ac_hysteresis_level</code>	hysteresis_level	Hysteresis level of the comparator channel
enum <code>ac_chan_interrupt_selection</code>	interrupt_selection	Interrupt criteria for the comparator channel, to select the condition that will trigger a callback
enum <code>ac_chan_neg_mux</code>	negative_input	Input multiplexer selection for the comparator's negative input pin. Any internal reference source, such as a bandgap reference voltage or the DAC, must be configured and enabled prior to its use as a comparator input.
enum <code>ac_chan_output</code>	output_mode	Output mode of the comparator, whether it should be available for internal use, or asynchronously/ synchronously linked to a general-purpose input/ output (GPIO) pin
enum <code>ac_chan_pos_mux</code>	positive_input	Input multiplexer selection for the comparator's positive input pin
bool	run_in_standby	If <code>true</code> , the comparator will continue to sample during sleep mode when triggered

Type	Name	Description
enum ac_chan_sample_mode	sample_mode	Sampling mode of the comparator channel
uint8_t	vcc_scale_factor	Scaled VCC voltage division factor for the channel, when a comparator pin is connected to the V _{CC} voltage scalar input. The formular is: $V_{scale} = V_{dd} * vcc\_scale\_factor / 64$. If the V _{CC} voltage scalar is not selected as a comparator channel pin's input, this value will be ignored.

2.6.2.2. Struct ac_config

Configuration structure for a comparator channel, to configure the input and output settings of the comparator.

Table 2-2. Members

Type	Name	Description
enum gclk_generator	source_generator	Source generator for AC GCLK

2.6.2.3. Struct ac_events

Event flags for the Analog Comparator module. This is used to enable and disable events via [ac_enable_events\(\)](#) and [ac_disable_events\(\)](#).

Table 2-3. Members

Type	Name	Description
bool	generate_event_on_state[]	If <code>true</code> , an event will be generated when a comparator state changes
bool	generate_event_on_window[]	If <code>true</code> , an event will be generated when a comparator window state changes
bool	on_event_sample[]	If <code>true</code> , a comparator will be sampled each time an event is received

2.6.2.4. Struct ac_module

AC software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

2.6.2.5. Struct ac_win_config

Table 2-4. Members

Type	Name	Description
enum ac_win_interrupt_selection	interrupt_selection	Interrupt criteria for the comparator window channel, to select the condition that will trigger a callback

2.6.3. Macro Definitions

2.6.3.1. Driver Feature Definition

Define AC driver feature set according to different device family.

Macro FEATURE_AC_HYSTERESIS_LEVEL

```
#define FEATURE_AC_HYSTERESIS_LEVEL
```

Setting of hysteresis level

Macro FEATURE_AC_SYNCBUSY_SCHEME_VERSION_2

```
#define FEATURE_AC_SYNCBUSY_SCHEME_VERSION_2
```

SYNCBUSY scheme version 2

Macro FEATURE_AC_RUN_IN_STANDY_EACH_COMPARATOR

```
#define FEATURE_AC_RUN_IN_STANDY_EACH_COMPARATOR
```

Run in standby feature for each comparator

2.6.3.2. AC Window Channel Status Flags

AC window channel status flags, returned by [ac_win_get_status\(\)](#).

Macro AC_WIN_STATUS_UNKNOWN

```
#define AC_WIN_STATUS_UNKNOWN
```

Unknown output state; the comparator window channel was not ready.

Macro AC_WIN_STATUS_ABOVE

```
#define AC_WIN_STATUS_ABOVE
```

Window Comparator's input voltage is above the window

Macro AC_WIN_STATUS_INSIDE

```
#define AC_WIN_STATUS_INSIDE
```

Window Comparator's input voltage is inside the window

Macro AC_WIN_STATUS_BELOW

```
#define AC_WIN_STATUS_BELOW
```

Window Comparator's input voltage is below the window

Macro AC_WIN_STATUS_INTERRUPT_SET

```
#define AC_WIN_STATUS_INTERRUPT_SET
```

This state reflects the window interrupt flag. When the interrupt flag should be set is configured in [ac_win_set_config\(\)](#). This state needs to be cleared by the of [ac_win_clear_status\(\)](#).

2.6.3.3. AC Channel Status Flags

AC channel status flags, returned by [ac_chan_get_status\(\)](#).

Macro AC_CHAN_STATUS_UNKNOWN

```
#define AC_CHAN_STATUS_UNKNOWN
```

Unknown output state; the comparator channel was not ready.

Macro AC_CHAN_STATUS_NEG_ABOVE_POS

```
#define AC_CHAN_STATUS_NEG_ABOVE_POS
```

Comparator's negative input pin is higher in voltage than the positive input pin.

Macro AC_CHAN_STATUS_POS_ABOVE_NEG

```
#define AC_CHAN_STATUS_POS_ABOVE_NEG
```

Comparator's positive input pin is higher in voltage than the negative input pin.

Macro AC_CHAN_STATUS_INTERRUPT_SET

```
#define AC_CHAN_STATUS_INTERRUPT_SET
```

This state reflects the channel interrupt flag. When the interrupt flag should be set is configured in [ac_chan_set_config\(\)](#). This state needs to be cleared by the of [ac_chan_clear_status\(\)](#).

2.6.4. Function Definitions

2.6.4.1. Configuration and Initialization

Function `ac_reset()`

Resets and disables the Analog Comparator driver.

```
enum status_code ac_reset(  
    struct ac_module *const module_inst)
```

Resets and disables the Analog Comparator driver, resets the internal states and registers of the hardware module to their power-on defaults.

Table 2-5. Parameters

Data direction	Parameter name	Description
[out]	module_inst	Pointer to the AC software instance struct

Function `ac_init()`

Initializes and configures the Analog Comparator driver.

```
enum status_code ac_init(  
    struct ac_module *const module_inst,  
    Ac *const hw,  
    struct ac_config *const config)
```

Initializes the Analog Comparator driver, configuring it to the user supplied configuration parameters, ready for use. This function should be called before enabling the Analog Comparator.

Note: Once called the Analog Comparator will not be running; to start the Analog Comparator call [ac_enable\(\)](#) after configuring the module.

Table 2-6. Parameters

Data direction	Parameter name	Description
[out]	module_inst	Pointer to the AC software instance struct
[in]	hw	Pointer to the AC module instance
[in]	config	Pointer to the config struct, created by the user application

Function ac_is_syncing()

Determines if the hardware module(s) are currently synchronizing to the bus.

```
bool ac_is_syncing(
    struct ac_module *const module_inst)
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Table 2-7. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the AC software instance struct

Returns

Synchronization status of the underlying hardware module(s).

Table 2-8. Return Values

Return value	Description
false	If the module has completed synchronization
true	If the module synchronization is ongoing

Function ac_get_config_defaults()

Initializes all members of an Analog Comparator configuration structure to safe defaults.

```
void ac_get_config_defaults(
    struct ac_config *const config)
```

Initializes all members of a given Analog Comparator configuration structure to safe known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- All comparator pairs disabled during sleep mode (if has this feature)
- Generator 0 is the default GCLK generator

Table 2-9. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `ac_enable()`

Enables an Analog Comparator that was previously configured.

```
void ac_enable(  
    struct ac_module *const module_inst)
```

Enables an Analog Comparator that was previously configured via a call to `ac_init()`.

Table 2-10. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral

Function `ac_disable()`

Disables an Analog Comparator that was previously enabled.

```
void ac_disable(  
    struct ac_module *const module_inst)
```

Disables an Analog Comparator that was previously started via a call to `ac_enable()`.

Table 2-11. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral

Function `ac_enable_events()`

Enables an Analog Comparator event input or output.

```
void ac_enable_events(  
    struct ac_module *const module_inst,  
    struct ac_events *const events)
```

Enables one or more input or output events to or from the Analog Comparator module. See `ac_events` for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 2-12. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	events	Struct containing flags of events to enable

Function `ac_disable_events()`

Disables an Analog Comparator event input or output.

```
void ac_disable_events(  
    struct ac_module *const module_inst,  
    struct ac_events *const events)
```

Disables one or more input or output events to or from the Analog Comparator module. See `ac_events` for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 2-13. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	events	Struct containing flags of events to disable

2.6.4.2. Channel Configuration and Initialization

Function `ac_chan_get_config_defaults()`

Initializes all members of an Analog Comparator channel configuration structure to safe defaults.

```
void ac_chan_get_config_defaults(  
    struct ac_chan_config *const config)
```

Initializes all members of an Analog Comparator channel configuration structure to safe defaults. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Continuous sampling mode
- Majority of five sample output filter
- Comparator disabled during sleep mode (if has this feature)
- Hysteresis enabled on the input pins
- Hysteresis level of 50mV if having this feature
- Internal comparator output mode
- Comparator pin multiplexer 0 selected as the positive input
- Scaled V_{CC} voltage selected as the negative input
- V_{CC} voltage scaler set for a division factor of two
- Channel interrupt set to occur when the compare threshold is passed

Table 2-14. Parameters

Data direction	Parameter name	Description
[out]	config	Channel configuration structure to initialize to default values

Function `ac_chan_set_config()`

Writes an Analog Comparator channel configuration to the hardware module.

```
enum status_code ac_chan_set_config(  
    struct ac_module *const module_inst,  
    const enum ac_chan_channel channel,  
    struct ac_chan_config *const config)
```

Writes a given Analog Comparator channel configuration to the hardware module.

Table 2-15. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Analog Comparator channel to configure
[in]	config	Pointer to the channel configuration struct

Function ac_chan_enable()

Enables an Analog Comparator channel that was previously configured.

```
void ac_chan_enable(
    struct ac_module *const module_inst,
    const enum ac_chan_channel channel)
```

Enables an Analog Comparator channel that was previously configured via a call to [ac_chan_set_config\(\)](#).

Table 2-16. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Comparator channel to enable

Function ac_chan_disable()

Disables an Analog Comparator channel that was previously enabled.

```
void ac_chan_disable(
    struct ac_module *const module_inst,
    const enum ac_chan_channel channel)
```

Stops an Analog Comparator channel that was previously started via a call to [ac_chan_enable\(\)](#).

Table 2-17. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Comparator channel to disable

2.6.4.3. Channel Control**Function ac_chan_trigger_single_shot()**

Triggers a comparison on a comparator that is configured in single shot mode.

```
void ac_chan_trigger_single_shot(
    struct ac_module *const module_inst,
    const enum ac_chan_channel channel)
```

Triggers a single conversion on a comparator configured to compare on demand (single shot mode) rather than continuously.

Table 2-18. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Comparator channel to trigger

Function `ac_chan_is_ready()`

Determines if a given comparator channel is ready for comparisons.

```
bool ac_chan_is_ready(
    struct ac_module *const module_inst,
    const enum ac_chan_channel channel)
```

Checks a comparator channel to see if the comparator is currently ready to begin comparisons.

Table 2-19. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Comparator channel to test

Returns

Comparator channel readiness state.

Function `ac_chan_get_status()`

Determines the output state of a comparator channel.

```
uint8_t ac_chan_get_status(
    struct ac_module *const module_inst,
    const enum ac_chan_channel channel)
```

Retrieves the last comparison value (after filtering) of a given comparator. If the comparator was not ready at the time of the check, the comparison result will be indicated as being unknown.

Table 2-20. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Comparator channel to test

Returns

Bit mask of comparator channel status flags.

Function `ac_chan_clear_status()`

Clears an interrupt status flag.

```
void ac_chan_clear_status(
    struct ac_module *const module_inst,
    const enum ac_chan_channel channel)
```

This function is used to clear the AC_CHAN_STATUS_INTERRUPT_SET flag it will clear the flag for the channel indicated by the channel argument.

Table 2-21. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	channel	Comparator channel to clear

2.6.4.4. Window Mode Configuration and Initialization

Function ac_win_get_config_defaults()

Initializes an Analog Comparator window configuration structure to defaults.

```
void ac_win_get_config_defaults(
    struct ac_win_config *const config)
```

Initializes a given Analog Comparator channel configuration structure to a set of known default values. This function should be called if window interrupts are needed and before [ac_win_set_config\(\)](#).

The default configuration is as follows:

- Channel interrupt set to occur when the measurement is above the window

Table 2-22. Parameters

Data direction	Parameter name	Description
[out]	config	Window configuration structure to initialize to default values

Function ac_win_set_config()

Function used to setup interrupt selection of a window.

```
enum status_code ac_win_set_config(
    struct ac_module *const module_inst,
    enum ac_win_channel const win_channel,
    struct ac_win_config *const config)
```

This function is used to setup when an interrupt should occur for a given window.

Note: This must be done before enabling the channel.

Table 2-23. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to software instance struct
[in]	win_channel	Window channel to setup
[in]	config	Configuration for the given window channel

Table 2-24. Return Values

Return value	Description
STATUS_OK	Function exited successful
STATUS_ERR_INVALID_ARG	win_channel argument incorrect

Function `ac_win_enable()`

Enables an Analog Comparator window channel that was previously configured.

```
enum status_code ac_win_enable(
    struct ac_module *const module_inst,
    const enum ac_win_channel win_channel)
```

Enables and starts an Analog Comparator window channel.

Note: The comparator channels used by the window channel must be configured and enabled before calling this function. The two comparator channels forming each window comparator pair must have identical configurations other than the negative pin multiplexer setting.

Table 2-25. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	win_channel	Comparator window channel to enable

Returns

Status of the window enable procedure.

Table 2-26. Return Values

Return value	Description
STATUS_OK	The window comparator was enabled
STATUS_ERR_IO	One or both comparators in the window comparator pair is disabled
STATUS_ERR_BAD_FORMAT	The comparator channels in the window pair were not configured correctly

Function `ac_win_disable()`

Disables an Analog Comparator window channel that was previously enabled.

```
void ac_win_disable(
    struct ac_module *const module_inst,
    const enum ac_win_channel win_channel)
```

Stops an Analog Comparator window channel that was previously started via a call to `ac_win_enable()`.

Table 2-27. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	win_channel	Comparator window channel to disable

2.6.4.5. Window Mode Control

Function ac_win_is_ready()

Determines if a given Window Comparator is ready for comparisons.

```
bool ac_win_is_ready(
    struct ac_module *const module_inst,
    const enum ac_win_channel win_channel)
```

Checks a Window Comparator to see if the both comparators used for window detection is currently ready to begin comparisons.

Table 2-28. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	win_channel	Window Comparator channel to test

Returns

Window Comparator channel readiness state.

Function ac_win_get_status()

Determines the state of a specified Window Comparator.

```
uint8_t ac_win_get_status(
    struct ac_module *const module_inst,
    const enum ac_win_channel win_channel)
```

Retrieves the current window detection state, indicating what the input signal is currently comparing to relative to the window boundaries.

Table 2-29. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Analog Comparator peripheral
[in]	win_channel	Comparator Window channel to test

Returns

Bit mask of Analog Comparator window channel status flags.

Function `ac_win_clear_status()`

Clears an interrupt status flag.

```
void ac_win_clear_status(  
    struct ac_module *const module_inst,  
    const enum ac_win_channel win_channel)
```

This function is used to clear the `AC_WIN_STATUS_INTERRUPT_SET` flag it will clear the flag for the channel indicated by the `win_channel` argument.

Table 2-30. Parameters

Data direction	Parameter name	Description
[in]	<code>module_inst</code>	Software instance for the Analog Comparator peripheral
[in]	<code>win_channel</code>	Window channel to clear

2.6.5. Enumeration Definitions

2.6.5.1. Enum `ac_callback`

Enum for possible callback types for the AC module.

Table 2-31. Members

Enum value	Description
<code>AC_CALLBACK_COMPARATOR_0</code>	Callback for comparator 0
<code>AC_CALLBACK_COMPARATOR_1</code>	Callback for comparator 1
<code>AC_CALLBACK_WINDOW_0</code>	Callback for window 0

2.6.5.2. Enum `ac_chan_channel`

Enum for the possible comparator channels.

Table 2-32. Members

Enum value	Description
<code>AC_CHAN_CHANNEL_0</code>	Comparator channel 0 (Pair 0, Comparator 0)
<code>AC_CHAN_CHANNEL_1</code>	Comparator channel 1 (Pair 0, Comparator 1)
<code>AC_CHAN_CHANNEL_2</code>	Comparator channel 2 (Pair 1, Comparator 0)
<code>AC_CHAN_CHANNEL_3</code>	Comparator channel 3 (Pair 1, Comparator 1)

2.6.5.3. Enum `ac_chan_filter`

Enum for the possible channel output filtering configurations of an Analog Comparator channel.

Table 2-33. Members

Enum value	Description
AC_CHAN_FILTER_NONE	No output filtering is performed on the comparator channel
AC_CHAN_FILTER_MAJORITY_3	Comparator channel output is passed through a Majority-of-Three filter
AC_CHAN_FILTER_MAJORITY_5	Comparator channel output is passed through a Majority-of-Five filter

2.6.5.4. Enum `ac_chan_interrupt_selection`

This enum is used to select when a channel interrupt should occur.

Table 2-34. Members

Enum value	Description
AC_CHAN_INTERRUPT_SELECTION_TOGGLE	An interrupt will be generated when the comparator level is passed
AC_CHAN_INTERRUPT_SELECTION_RISING	An interrupt will be generated when the measurement goes above the compare level
AC_CHAN_INTERRUPT_SELECTION_FALLING	An interrupt will be generated when the measurement goes below the compare level
AC_CHAN_INTERRUPT_SELECTION_END_OF_COMPARE	An interrupt will be generated when a new measurement is complete. Interrupts will only be generated in single shot mode. This state needs to be cleared by the use of <code>ac_chan_clear_status()</code>

2.6.5.5. Enum `ac_chan_neg_mux`

Enum for the possible channel negative pin input of an Analog Comparator channel.

Table 2-35. Members

Enum value	Description
AC_CHAN_NEG_MUX_PIN0	Negative comparator input is connected to physical AC input pin 0
AC_CHAN_NEG_MUX_PIN1	Negative comparator input is connected to physical AC input pin 1
AC_CHAN_NEG_MUX_PIN2	Negative comparator input is connected to physical AC input pin 2
AC_CHAN_NEG_MUX_PIN3	Negative comparator input is connected to physical AC input pin 3

Enum value	Description
AC_CHAN_NEG_MUX_GND	Negative comparator input is connected to the internal ground plane
AC_CHAN_NEG_MUX_SCALED_VCC	Negative comparator input is connected to the channel's internal V_{CC} plane voltage scalar
AC_CHAN_NEG_MUX_BANDGAP	Negative comparator input is connected to the internal band gap voltage reference
AC_CHAN_NEG_MUX_DAC0	For SAM D20/D21/D10/D11/R21/DA1: Negative comparator input is connected to the channel's internal DAC channel 0 output. For SAM L21/C20/C21: Negative comparator input is connected to the channel's internal DAC channel 0 output for Comparator 0 or OPAMP output for Comparator 1.

2.6.5.6. Enum `ac_chan_output`

Enum for the possible channel GPIO output routing configurations of an Analog Comparator channel.

Table 2-36. Members

Enum value	Description
AC_CHAN_OUTPUT_INTERNAL	Comparator channel output is not routed to a physical GPIO pin, and is used internally only
AC_CHAN_OUTPUT_ASYNCRONOUS	Comparator channel output is routed to its matching physical GPIO pin, via an asynchronous path
AC_CHAN_OUTPUT_SYNCHRONOUS	Comparator channel output is routed to its matching physical GPIO pin, via a synchronous path

2.6.5.7. Enum `ac_chan_pos_mux`

Enum for the possible channel positive pin input of an Analog Comparator channel.

Table 2-37. Members

Enum value	Description
AC_CHAN_POS_MUX_PIN0	Positive comparator input is connected to physical AC input pin 0
AC_CHAN_POS_MUX_PIN1	Positive comparator input is connected to physical AC input pin 1
AC_CHAN_POS_MUX_PIN2	Positive comparator input is connected to physical AC input pin 2
AC_CHAN_POS_MUX_PIN3	Positive comparator input is connected to physical AC input pin 3

2.6.5.8. Enum `ac_chan_sample_mode`

Enum for the possible channel sampling modes of an Analog Comparator channel.

Table 2-38. Members

Enum value	Description
AC_CHAN_MODE_CONTINUOUS	Continuous sampling mode; when the channel is enabled the comparator output is available for reading at any time
AC_CHAN_MODE_SINGLE_SHOT	Single shot mode; when used the comparator channel must be triggered to perform a comparison before reading the result

2.6.5.9. Enum ac_hysteresis_level

Enum for possible hysteresis level types for AC module.

Table 2-39. Members

Enum value	Description
AC_HYSTERESIS_LEVEL_50	Hysteresis level of 50mV
AC_HYSTERESIS_LEVEL_70	Hysteresis level of 70mV
AC_HYSTERESIS_LEVEL_90	Hysteresis level of 90mV
AC_HYSTERESIS_LEVEL_110	Hysteresis level of 110mV

2.6.5.10. Enum ac_win_channel

Enum for the possible window comparator channels.

Table 2-40. Members

Enum value	Description
AC_WIN_CHANNEL_0	Window channel 0 (Pair 0, Comparators 0 and 1)
AC_WIN_CHANNEL_1	Window channel 1 (Pair 1, Comparators 2 and 3)

2.6.5.11. Enum ac_win_interrupt_selection

This enum is used to select when a window interrupt should occur.

Table 2-41. Members

Enum value	Description
AC_WIN_INTERRUPT_SELECTION_ABOVE	Interrupt is generated when the compare value goes above the window
AC_WIN_INTERRUPT_SELECTION_INSIDE	Interrupt is generated when the compare value goes inside the window
AC_WIN_INTERRUPT_SELECTION_BELOW	Interrupt is generated when the compare value goes below the window
AC_WIN_INTERRUPT_SELECTION_OUTSIDE	Interrupt is generated when the compare value goes outside the window

2.7. Extra Information for AC Driver

2.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
AC	Analog Comparator
DAC	Digital-to-Analog Converter
MUX	Multiplexer

2.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

2.7.3. Errata

There are no errata related to this driver.

2.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

2.8. Examples for AC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Analog Comparator \(AC\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for AC - Basic](#)
- [Quick Start Guide for AC - Callback](#)

2.8.1. Quick Start Guide for AC - Basic

In this use case, the Analog Comparator module is configured for:

- Comparator peripheral in manually triggered (e.g. "Single Shot" mode)
- One comparator channel connected to input MUX pin 0 and compared to a scaled $V_{CC}/2$ voltage

This use case sets up the Analog Comparator to compare an input voltage fed into a GPIO pin of the device against a scaled voltage of the microcontroller's V_{CC} power rail. The comparisons are made on-demand in single-shot mode, and the result stored into a local variable which is then output to the board LED to visually show the comparison state.

2.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
/* AC module software instance (must not go out of scope while in use) */
static struct ac_module ac_instance;

/* Comparator channel that will be used */
#define AC_COMPARATOR_CHANNEL    AC_CHAN_CHANNEL_0

void configure_ac(void)
{
    /* Create a new configuration structure for the Analog Comparator settings
    * and fill with the default module settings. */
    struct ac_config config_ac;
    ac_get_config_defaults(&config_ac);

    /* Alter any Analog Comparator configuration settings here if required */

    /* Initialize and enable the Analog Comparator with the user settings */
    ac_init(&ac_instance, AC, &config_ac);
}

void configure_ac_channel(void)
{
    /* Create a new configuration structure for the Analog Comparator channel
    * settings and fill with the default module channel settings. */
    struct ac_chan_config ac_chan_conf;
    ac_chan_get_config_defaults(&ac_chan_conf);

    /* Set the Analog Comparator channel configuration settings */
    ac_chan_conf.sample_mode      = AC_CHAN_MODE_SINGLE_SHOT;
    ac_chan_conf.positive_input   = AC_CHAN_POS_MUX_PIN0;
    ac_chan_conf.negative_input   = AC_CHAN_NEG_MUX_SCALED_VCC;
    ac_chan_conf.vcc_scale_factor = 32;

    /* Set up a pin as an AC channel input */
    struct system_pinmux_config ac0_pin_conf;
    system_pinmux_get_config_defaults(&ac0_pin_conf);
    ac0_pin_conf.direction        = SYSTEM_PINMUX_PIN_DIR_INPUT;
    ac0_pin_conf.mux_position     = CONF_AC_MUX;
    system_pinmux_pin_set_config(CONF_AC_PIN, &ac0_pin_conf);

    /* Initialize and enable the Analog Comparator channel with the user
    * settings */
    ac_chan_set_config(&ac_instance, AC_COMPARATOR_CHANNEL, &ac_chan_conf);
    ac_chan_enable(&ac_instance, AC_COMPARATOR_CHANNEL);
}
```

Add to user application initialization (typically the start of `main()`):

```
system_init();
configure_ac();
configure_ac_channel();
ac_enable(&ac_instance);
```


Workflow

1. Create an AC device instance struct, which will be associated with an Analog Comparator peripheral hardware instance.

```
static struct ac_module ac_instance;
```

Note: Device instance structures shall never go out of scope when in use.

2. Define a macro to select the comparator channel that will be sampled, for convenience.

```
#define AC_COMPARATOR_CHANNEL    AC_CHAN_CHANNEL_0
```

3. Create a new function `configure_ac()`, which will be used to configure the overall Analog Comparator peripheral.

```
void configure_ac(void)
```

4. Create an Analog Comparator peripheral configuration structure that will be filled out to set the module configuration.

```
struct ac_config config_ac;
```

5. Fill the Analog Comparator peripheral configuration structure with the default module configuration values.

```
ac_get_config_defaults(&config_ac);
```

6. Initialize the Analog Comparator peripheral and associate it with the software instance structure that was defined previously.

```
ac_init(&ac_instance, AC, &config_ac);
```

7. Create a new function `configure_ac_channel()`, which will be used to configure the overall Analog Comparator peripheral.

```
void configure_ac_channel(void)
```

8. Create an Analog Comparator channel configuration structure that will be filled out to set the channel configuration.

```
struct ac_chan_config ac_chan_conf;
```

9. Fill the Analog Comparator channel configuration structure with the default channel configuration values.

```
ac_chan_get_config_defaults(&ac_chan_conf);
```

10. Alter the channel configuration parameters to set the channel to one-shot mode, with the correct negative and positive MUX selections and the desired voltage scaler.

```
ac_chan_conf.sample_mode      = AC_CHAN_MODE_SINGLE_SHOT;  
ac_chan_conf.positive_input   = AC_CHAN_POS_MUX_PIN0;  
ac_chan_conf.negative_input   = AC_CHAN_NEG_MUX_SCALED_VCC;  
ac_chan_conf.vcc_scale_factor = 32;
```

Note: The voltage scalar formula is documented in description for [ac_chan_config::vcc_scale_factor](#).

11. Configure the physical pin that will be routed to the AC module channel 0.

```
struct system_pinmux_config ac0_pin_conf;  
system_pinmux_get_config_defaults(&ac0_pin_conf);  
ac0_pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
```

```
ac0_pin_conf.mux_position = CONF_AC_MUX;
system_pinmux_pin_set_config(CONF_AC_PIN, &ac0_pin_conf);
```

12. Initialize the Analog Comparator channel and configure it with the desired settings.

```
ac_chan_set_config(&ac_instance, AC_COMPARATOR_CHANNEL, &ac_chan_conf);
```

13. Enable the now initialized Analog Comparator channel.

```
ac_chan_enable(&ac_instance, AC_COMPARATOR_CHANNEL);
```

14. Enable the now initialized Analog Comparator peripheral.

```
ac_enable(&ac_instance);
```

2.8.1.2. Implementation

Code

Copy-paste the following code to your user application:

```
ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
uint8_t last_comparison = AC_CHAN_STATUS_UNKNOWN;
while (true) {
    if (ac_chan_is_ready(&ac_instance, AC_COMPARATOR_CHANNEL)) {
        do {
            last_comparison = ac_chan_get_status(&ac_instance,
                                                AC_COMPARATOR_CHANNEL);
        } while (last_comparison & AC_CHAN_STATUS_UNKNOWN);

        port_pin_set_output_level(LED_0_PIN,
                                   (last_comparison & AC_CHAN_STATUS_NEG_ABOVE_POS));

        ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
    }
}
```

Workflow

1. Trigger the first comparison on the comparator channel.

```
ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
```

2. Create a local variable to maintain the current comparator state. Since no comparison has taken place, it is initialized to [AC_CHAN_STATUS_UNKNOWN](#).

```
uint8_t last_comparison = AC_CHAN_STATUS_UNKNOWN;
```

3. Make the application loop infinitely, while performing triggered comparisons.

```
while (true) {
```

4. Check if the comparator is ready for the last triggered comparison result to be read.

```
if (ac_chan_is_ready(&ac_instance, AC_COMPARATOR_CHANNEL)) {
```

5. Read the comparator output state into the local variable for application use, re-trying until the comparison state is ready.

```
do {
    last_comparison = ac_chan_get_status(&ac_instance,
                                        AC_COMPARATOR_CHANNEL);
} while (last_comparison & AC_CHAN_STATUS_UNKNOWN);
```

6. Set the board LED state to mirror the last comparison state.

```
port_pin_set_output_level(LED_0_PIN,  
    (last_comparison & AC_CHAN_STATUS_NEG_ABOVE_POS));
```

7. Trigger the next conversion on the Analog Comparator channel.

```
ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
```

2.8.2. Quick Start Guide for AC - Callback

In this use case, the Analog Comparator module is configured for:

- Comparator peripheral in manually triggered (e.g. "Single Shot" mode)
- One comparator channel connected to input MUX pin 0 and compared to a scaled $V_{CC}/2$ voltage

This use case sets up the Analog Comparator to compare an input voltage fed into a GPIO pin of the device against a scaled voltage of the microcontroller's V_{CC} power rail. The comparisons are made on-demand in single-shot mode, and the result stored into a local variable which is then output to the board LED to visually show the comparison state.

2.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
/* AC module software instance (must not go out of scope while in use). */  
static struct ac_module ac_instance;  
  
/* Comparator channel that will be used. */  
#define AC_COMPARATOR_CHANNEL    AC_CHAN_CHANNEL_0  
  
void configure_ac(void)  
{  
    /* Create a new configuration structure for the Analog Comparator settings  
    * and fill with the default module settings. */  
    struct ac_config config_ac;  
    ac_get_config_defaults(&config_ac);  
  
    /* Alter any Analog Comparator configuration settings here if required. */  
  
    /* Initialize and enable the Analog Comparator with the user settings. */  
    ac_init(&ac_instance, AC, &config_ac);  
}  
  
void configure_ac_channel(void)  
{  
    /* Create a new configuration structure for the Analog Comparator channel  
    * settings and fill with the default module channel settings. */  
    struct ac_chan_config config_ac_chan;  
    ac_chan_get_config_defaults(&config_ac_chan);  
  
    /* Set the Analog Comparator channel configuration settings. */  
    config_ac_chan.sample_mode      = AC_CHAN_MODE_SINGLE_SHOT;  
    config_ac_chan.positive_input   = AC_CHAN_POS_MUX_PIN0;  
    config_ac_chan.negative_input   = AC_CHAN_NEG_MUX_SCALED_VCC;  
    config_ac_chan.vcc_scale_factor = 32;  
    config_ac_chan.interrupt_selection =  
    AC_CHAN_INTERRUPT_SELECTION_END_OF_COMPARE;
```

```

/* Set up a pin as an AC channel input. */
struct system_pinmux_config ac0_pin_conf;
system_pinmux_get_config_defaults(&ac0_pin_conf);
ac0_pin_conf.direction      = SYSTEM_PINMUX_PIN_DIR_INPUT;
ac0_pin_conf.mux_position = CONF_AC_MUX;
system_pinmux_pin_set_config(CONF_AC_PIN, &ac0_pin_conf);

/* Initialize and enable the Analog Comparator channel with the user
 * settings. */
ac_chan_set_config(&ac_instance, AC_COMPARATOR_CHANNEL,
&config_ac_chan);
ac_chan_enable(&ac_instance, AC_COMPARATOR_CHANNEL);
}

void callback_function_ac(struct ac_module *const module_inst)
{
    callback_status = true;
}

void configure_ac_callback(void)
{
    ac_register_callback(&ac_instance, callback_function_ac,
AC_CALLBACK_COMPARATOR_0);
    ac_enable_callback(&ac_instance, AC_CALLBACK_COMPARATOR_0);
}

```

Add to user application initialization (typically the start of `main()`):

```

system_init();
configure_ac();
configure_ac_channel();
configure_ac_callback();

ac_enable(&ac_instance);

```

Workflow

1. Create an AC device instance struct, which will be associated with an Analog Comparator peripheral hardware instance.

```
static struct ac_module ac_instance;
```

Note: Device instance structures shall never go out of scope when in use.

2. Define a macro to select the comparator channel that will be sampled, for convenience.

```
#define AC_COMPARATOR_CHANNEL    AC_CHAN_CHANNEL_0
```

3. Create a new function `configure_ac()`, which will be used to configure the overall Analog Comparator peripheral.

```
void configure_ac(void)
{
```

4. Create an Analog Comparator peripheral configuration structure that will be filled out to set the module configuration.

```
struct ac_config config_ac;
```

5. Fill the Analog Comparator peripheral configuration structure with the default module configuration values.

```
ac_get_config_defaults(&config_ac);
```

6. Initialize the Analog Comparator peripheral and associate it with the software instance structure that was defined previously.

```
ac_init(&ac_instance, AC, &config_ac);
```

7. Create a new function `configure_ac_channel()`, which will be used to configure the overall Analog Comparator peripheral.

```
void configure_ac_channel(void)
{
```

8. Create an Analog Comparator channel configuration structure that will be filled out to set the channel configuration.

```
struct ac_chan_config config_ac_chan;
```

9. Fill the Analog Comparator channel configuration structure with the default channel configuration values.

```
ac_chan_get_config_defaults(&config_ac_chan);
```

10. Alter the channel configuration parameters to set the channel to one-shot mode, with the correct negative and positive MUX selections and the desired voltage scaler.

Note: The voltage scalar formula is documented in description for [ac_chan_config::vcc_scale_factor](#).

11. Select when the interrupt should occur. In this case an interrupt will occur at every finished conversion.

```
config_ac_chan.sample_mode          = AC_CHAN_MODE_SINGLE_SHOT;
config_ac_chan.positive_input       = AC_CHAN_POS_MUX_PIN0;
config_ac_chan.negative_input       = AC_CHAN_NEG_MUX_SCALED_VCC;
config_ac_chan.vcc_scale_factor     = 32;
config_ac_chan.interrupt_selection = AC_CHAN_INTERRUPT_SELECTION_END_OF_COMPARE;
```

12. Configure the physical pin that will be routed to the AC module channel 0.

```
struct system_pinmux_config ac0_pin_conf;
system_pinmux_get_config_defaults(&ac0_pin_conf);
ac0_pin_conf.direction      = SYSTEM_PINMUX_PIN_DIR_INPUT;
ac0_pin_conf.mux_position   = CONF_AC_MUX;
system_pinmux_pin_set_config(CONF_AC_PIN, &ac0_pin_conf);
```

13. Initialize the Analog Comparator channel and configure it with the desired settings.

```
ac_chan_set_config(&ac_instance, AC_COMPARATOR_CHANNEL,
&config_ac_chan);
```

14. Enable the initialized Analog Comparator channel.

```
ac_chan_enable(&ac_instance, AC_COMPARATOR_CHANNEL);
```

15. Create a new callback function.

```
void callback_function_ac(struct ac_module *const module_inst)
{
    callback_status = true;
}
```

16. Create a callback status software flag.

```
bool volatile callback_status = false;
```

17. Let the callback function set the callback_status flag to true.

```
callback_status = true;
```

18. Create a new function `configure_ac_callback()`, which will be used to configure the callbacks.

```
void configure_ac_callback(void)
{
    ac_register_callback(&ac_instance, callback_function_ac,
AC_CALLBACK_COMPARATOR_0);
    ac_enable_callback(&ac_instance, AC_CALLBACK_COMPARATOR_0);
}
```

19. Register callback function.

```
ac_register_callback(&ac_instance, callback_function_ac,
AC_CALLBACK_COMPARATOR_0);
```

20. Enable the callbacks.

```
ac_enable_callback(&ac_instance, AC_CALLBACK_COMPARATOR_0);
```

21. Enable the now initialized Analog Comparator peripheral.

```
ac_enable(&ac_instance);
```

Note: This should not be done until after the AC is setup and ready to be used.

2.8.2.2. Implementation

Code

Copy-paste the following code to your user application:

```
ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);

uint8_t last_comparison = AC_CHAN_STATUS_UNKNOWN;
port_pin_set_output_level(LED_0_PIN, true);
while (true) {
    if (callback_status == true) {
        do
        {
            last_comparison = ac_chan_get_status(&ac_instance,
AC_COMPARATOR_CHANNEL);
        } while (last_comparison & AC_CHAN_STATUS_UNKNOWN);
        port_pin_set_output_level(LED_0_PIN,
            (last_comparison & AC_CHAN_STATUS_NEG_ABOVE_POS));
        callback_status = false;
        ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
    }
}
```

Workflow

1. Trigger the first comparison on the comparator channel.

```
ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
```

2. Create a local variable to maintain the current comparator state. Since no comparison has taken place, it is initialized to `AC_CHAN_STATUS_UNKNOWN`.

```
uint8_t last_comparison = AC_CHAN_STATUS_UNKNOWN;
```

3. Make the application loop infinitely, while performing triggered comparisons.

```
while (true) {
```

4. Check if a new comparison is complete.

```
if (callback_status == true) {
```

5. Check if the comparator is ready for the last triggered comparison result to be read.

```
do
{
    last_comparison = ac_chan_get_status(&ac_instance,
        AC_COMPARATOR_CHANNEL);
} while (last_comparison & AC_CHAN_STATUS_UNKNOWN);
```

6. Read the comparator output state into the local variable for application use, re-trying until the comparison state is ready.

```
do
{
    last_comparison = ac_chan_get_status(&ac_instance,
        AC_COMPARATOR_CHANNEL);
} while (last_comparison & AC_CHAN_STATUS_UNKNOWN);
```

7. Set the board LED state to mirror the last comparison state.

```
port_pin_set_output_level(LED_0_PIN,
    (last_comparison & AC_CHAN_STATUS_NEG_ABOVE_POS));
```

8. After the interrupt is handled, set the software callback flag to false.

```
callback_status = false;
```

9. Trigger the next conversion on the Analog Comparator channel.

```
ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
```

3. SAM Analog-to-Digital Converter (ADC) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Analog-to-Digital Converter functionality, for the conversion of analog voltages into a corresponding digital form. The following driver Application Programming Interface (API) modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- ADC (Analog-to-Digital Converter)

The following devices can use this module:

- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

3.1. Prerequisites

There are no prerequisites for this module.

3.2. Module Overview

This driver provides an interface for the Analog-to-Digital conversion functions on the device, to convert analog voltages to a corresponding digital value. The ADC has up to 12-bit resolution, and is capable of converting up to 1,000,000 samples per second (MSPS).

The ADC has a compare function for accurate monitoring of user defined thresholds with minimum software intervention required. The ADC may be configured for 8-, 10-, or 12-bit result, reducing the conversion time. ADC conversion results are provided left or right adjusted which eases calculation when the result is represented as a signed integer.

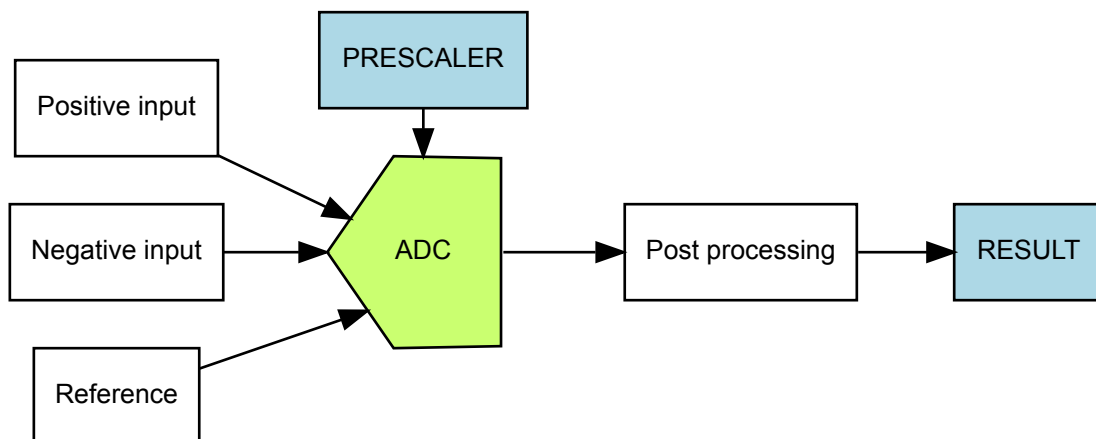
The input selection is flexible, and both single-ended and differential measurements can be made. For differential measurements, an optional gain stage is available to increase the dynamic range. In addition, several internal signal inputs are available. The ADC can provide both signed and unsigned results.

The ADC measurements can either be started by application software or an incoming event from another peripheral in the device, and both internal and external reference voltages can be selected.

Note: Internal references will be enabled by the driver, but not disabled. Any reference not used by the application should be disabled by the application.

A simplified block diagram of the ADC can be seen in [Figure 3-1](#).

Figure 3-1. Module Overview



3.2.1. Sample Clock Prescaler

The ADC features a prescaler, which enables conversion at lower clock rates than the input Generic Clock to the ADC module. This feature can be used to lower the synchronization time of the digital interface to the ADC module via a high speed Generic Clock frequency, while still allowing the ADC sampling rate to be reduced.

3.2.2. ADC Resolution

The ADC supports full 8-, 10-, or 12-bit resolution. Hardware oversampling and decimation can be used to increase the effective resolution at the expense of throughput. Using oversampling and decimation mode the ADC resolution is increased from 12-bit to an effective 13-, 14-, 15-, or 16-bit. In these modes the conversion rate is reduced, as a greater number of samples is used to achieve the increased resolution. The available resolutions and effective conversion rate is listed in [Table 3-1](#).

Table 3-1. Effective ADC Conversion Speed Using Oversampling

Resolution	Effective conversion rate
13-bit	Conversion rate divided by 4
14-bit	Conversion rate divided by 16
15-bit	Conversion rate divided by 64
16-bit	Conversion rate divided by 256

3.2.3. Conversion Modes

ADC conversions can be software triggered on demand by the user application, if continuous sampling is not required. It is also possible to configure the ADC in free running mode, where new conversions are started as soon as the previous conversion is completed, or configure the ADC to scan across a number of input pins (see [Pin Scan](#)).

3.2.4. Differential and Single-ended Conversion

The ADC has two conversion modes; differential and single-ended. When measuring signals where the positive input pin is always at a higher voltage than the negative input pin, the single-ended conversion mode should be used in order to achieve a full 12-bit output resolution.

If however the positive input pin voltage may drop below the negative input pin the signed differential mode should be used.

3.2.5. Sample Time

The sample time for each ADC conversion is configurable as a number of half prescaled ADC clock cycles (depending on the prescaler value), allowing the user application to achieve faster or slower sampling depending on the source impedance of the ADC input channels. For applications with high impedance inputs the sample time can be increased to give the ADC an adequate time to sample and convert the input channel.

The resulting sampling time is given by the following equation:

$$t_{SAMPLE} = (sample\_length + 1) \times \frac{ADC_{CLK}}{2}$$

3.2.6. Averaging

The ADC can be configured to trade conversion speed for accuracy by averaging multiple samples in hardware. This feature is suitable when operating in noisy conditions.

You can specify any number of samples to accumulate (up to 1024) and the divide ratio to use (up to divide by 128). To modify these settings the `ADC_RESOLUTION_CUSTOM` needs to be set as the resolution. When this is set the number of samples to accumulate and the division ratio can be set by the configuration struct members `adc_config::accumulate_samples` and `adc_config::divide_result`. When using this mode the ADC result register will be set to be 16-bit wide to accommodate the larger result sizes produced by the accumulator.

The effective ADC conversion rate will be reduced by a factor of the number of accumulated samples; however, the effective resolution will be increased according to [Table 3-2](#).

Table 3-2. Effective ADC Resolution From Various Hardware Averaging Modes

Number of samples	Final result
1	12-bit
2	13-bit
4	14-bit
8	15-bit
16	16-bit
32	16-bit
64	16-bit
128	16-bit
256	16-bit
512	16-bit
1024	16-bit

3.2.7. Offset and Gain Correction

Inherent gain and offset errors affect the absolute accuracy of the ADC.

The offset error is defined as the deviation of the ADC's actual transfer function from ideal straight line at zero input voltage.

The gain error is defined as the deviation of the last output step's midpoint from the ideal straight line, after compensating for offset error.

The offset correction value is subtracted from the converted data before the result is ready. The gain correction value is multiplied with the offset corrected value.

The equation for both offset and gain error compensation is shown below:

$$ADC_{RESULT} = (VALUE_{CONV} + CORR_{OFFSET}) \times CORR_{GAIN}$$

When enabled, a given set of offset and gain correction values can be applied to the sampled data in hardware, giving a corrected stream of sample data to the user application at the cost of an increased sample latency.

In single conversion, a latency of 13 ADC Generic Clock cycles is added for the final sample result availability. As the correction time is always less than the propagation delay, in free running mode this latency appears only during the first conversion. After the first conversion is complete, future conversion results are available at the defined sampling rate.

3.2.8. Pin Scan

In pin scan mode, the first ADC conversion will begin from the configured positive channel, plus the requested starting offset. When the first conversion is completed, the next conversion will start at the next positive input channel and so on, until all requested pins to scan have been sampled and converted. SAM L21/L22 has automatic sequences feature instead of pin scan mode. In automatic sequence mode, all of 32 positives inputs can be included in a sequence. The sequence starts from the lowest input, and go to the next enabled input automatically.

Pin scanning gives a simple mechanism to sample a large number of physical input channel samples, using a single physical ADC channel.

3.2.9. Window Monitor

The ADC module window monitor function can be used to automatically compare the conversion result against a preconfigured pair of upper and lower threshold values.

The threshold values are evaluated differently, depending on whether differential or single-ended mode is selected. In differential mode, the upper and lower thresholds are evaluated as signed values for the comparison, while in single-ended mode the comparisons are made as a set of unsigned values.

The significant bits of the lower window monitor threshold and upper window monitor threshold values are user-configurable, and follow the overall ADC sampling bit precision set when the ADC is configured by the user application. For example, only the eight lower bits of the window threshold values will be compared to the sampled data whilst the ADC is configured in 8-bit mode. In addition, if using differential mode, the 8th bit will be considered as the sign bit even if bit 9 is zero.

3.2.10. Events

Event generation and event actions are configurable in the ADC.

The ADC has two actions that can be triggered upon event reception:

- Start conversion
- Flush pipeline and start conversion

The ADC can generate two events:

- Window monitor
- Result ready

If the event actions are enabled in the configuration, any incoming event will trigger the action.

If the window monitor event is enabled, an event will be generated when the configured window condition is detected.

If the result ready event is enabled, an event will be generated when a conversion is completed.

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the input event of another. For more information on event routing, refer to the event driver documentation.

3.3. Special Considerations

An integrated analog temperature sensor is available for use with the ADC. The bandgap voltage, as well as the scaled I/O and core voltages can also be measured by the ADC. For internal ADC inputs, the internal source(s) may need to be manually enabled by the user application before they can be measured.

3.4. Extra Information

For extra information, see [Extra Information for ADC Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

3.5. Examples

For a list of examples related to this driver, see [Examples for ADC Driver](#).

3.6. API Overview

3.6.1. Variable and Type Definitions

3.6.1.1. Type `adc_callback_t`

```
typedef void(* adc_callback_t )(struct adc_module *const module)
```

Type of the callback functions.

3.6.2. Structure Definitions

3.6.2.1. Struct `adc_config`

Configuration structure for an ADC instance. This structure should be initialized by the `adc_get_config_defaults()` function before being modified by the user application.

Table 3-3. Members

Type	Name	Description
enum <code>adc_accumulate_samples</code>	<code>accumulate_samples</code>	Number of ADC samples to accumulate when using the <code>ADC_RESOLUTION_CUSTOM</code> mode
enum <code>adc_clock_prescaler</code>	<code>clock_prescaler</code>	Clock prescaler
enum <code>gclk_generator</code>	<code>clock_source</code>	GCLK generator used to clock the peripheral
struct <code>adc_correction_config</code>	<code>correction</code>	Gain and offset correction configuration structure
bool	<code>differential_mode</code>	Enables differential mode if true
enum <code>adc_divide_result</code>	<code>divide_result</code>	Division ration when using the <code>ADC_RESOLUTION_CUSTOM</code> mode
enum <code>adc_event_action</code>	<code>event_action</code>	Event action to take on incoming event
bool	<code>freerunning</code>	Enables free running mode if true
bool	<code>left_adjust</code>	Left adjusted result
enum <code>adc_negative_input</code>	<code>negative_input</code>	Negative MUX input
bool	<code>on_demand</code>	ADC On demand control
enum <code>adc_positive_input</code>	<code>positive_input</code>	Positive MUX input
uint32_t	<code>positive_input_sequence_mask_enable</code>	Positive input enabled mask for conversion sequence. The sequence start from the lowest input, and go to the next enabled input automatically when the conversion is done. If no bits are set the sequence is disabled.
enum <code>adc_reference</code>	<code>reference</code>	Voltage reference

Type	Name	Description
bool	reference_compensation_enable	Enables reference buffer offset compensation if true. This will increase the accuracy of the gain stage, but decreases the input impedance; therefore the startup time of the reference must be increased.
enum adc_resolution	resolution	Result resolution
bool	run_in_standby	ADC run in standby control
uint8_t	sample_length	This value (0-63) control the ADC sampling time in number of half ADC prescaled clock cycles (depends of ADC_PRESCALER value), thus controlling the ADC input impedance. Sampling time is set according to the formula: Sample time = (sample_length+1) * (ADCclk / 2).
bool	sampling_time_compensation_enable	Enables sampling period offset compensation if true
struct adc_window_config	window	Window monitor configuration structure

3.6.2.2. Struct `adc_correction_config`

Gain and offset correction configuration structure. Part of the [adc_config](#) struct and will be initialized by [adc_get_config_defaults](#).

Table 3-4. Members

Type	Name	Description
bool	correction_enable	Enables correction for gain and offset based on values of <code>gain_correction</code> and <code>offset_correction</code> if set to true
uint16_t	gain_correction	This value defines how the ADC conversion result is compensated for gain error before written to the result register. This is a fractional value, 1-bit integer plus an 11-bit fraction, therefore $1/2 \leq \text{gain\_correction} < 2$. Valid <code>gain_correction</code> values ranges from <code>0b010000000000</code> to <code>0b111111111111</code> .
int16_t	offset_correction	This value defines how the ADC conversion result is compensated for offset error before written to the result register. This is a 12-bit value in two's complement format.

3.6.2.3. Struct `adc_events`

Event flags for the ADC module. This is used to enable and disable events via `adc_enable_events()` and `adc_disable_events()`.

Table 3-5. Members

Type	Name	Description
bool	<code>generate_event_on_conversion_done</code>	Enable event generation on conversion done
bool	<code>generate_event_on_window_monitor</code>	Enable event generation on window monitor

3.6.2.4. Struct `adc_module`

ADC software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

3.6.2.5. Struct `adc_window_config`

Window monitor configuration structure.

Table 3-6. Members

Type	Name	Description
int32_t	<code>window_lower_value</code>	Lower window value
enum <code>adc_window_mode</code>	<code>window_mode</code>	Selected window mode
int32_t	<code>window_upper_value</code>	Upper window value

3.6.3. Macro Definitions

3.6.3.1. Module Status Flags

ADC status flags, returned by `adc_get_status()` and cleared by `adc_clear_status()`.

Macro `ADC_STATUS_RESULT_READY`

```
#define ADC_STATUS_RESULT_READY
```

ADC result ready.

Macro `ADC_STATUS_WINDOW`

```
#define ADC_STATUS_WINDOW
```

Window monitor match.

Macro `ADC_STATUS_OVERRUN`

```
#define ADC_STATUS_OVERRUN
```

ADC result overwritten before read.

3.6.3.2. Macro `FEATURE_ADC_SUPPORT_MASTER_SLAVE`

```
#define FEATURE_ADC_SUPPORT_MASTER_SLAVE
```

Output Driver Strength Selection feature support.

3.6.4. Function Definitions

3.6.4.1. Driver Initialization and Configuration

Function `adc_init()`

Initializes the ADC.

```
enum status_code adc_init(  
    struct adc_module *const module_inst,  
    Adc * hw,  
    struct adc_config * config)
```

Initializes the ADC device struct and the hardware module based on the given configuration struct values.

Table 3-7. Parameters

Data direction	Parameter name	Description
[out]	module_inst	Pointer to the ADC software instance struct
[in]	hw	Pointer to the ADC module instance
[in]	config	Pointer to the configuration struct

Returns

Status of the initialization procedure.

Table 3-8. Return Values

Return value	Description
STATUS_OK	The initialization was successful
STATUS_ERR_INVALID_ARG	Invalid argument(s) were provided
STATUS_BUSY	The module is busy with a reset operation
STATUS_ERR_DENIED	The module is enabled

Function `adc_get_config_defaults()`

Initializes an ADC configuration structure to defaults.

```
void adc_get_config_defaults(  
    struct adc_config *const config)
```

Initializes a given ADC configuration struct to a set of known default values. This function should be called on any new instance of the configuration struct before being modified by the user application.

The default configuration is as follows:

- GCLK generator 0 (GCLK main) clock source
- Internal bandgap reference
- Div 2 clock prescaler
- 12-bit resolution
- Window monitor disabled

- Positive input on ADC PIN 1
- Negative input on Internal ground
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running disabled
- All events (input and generation) disabled
- ADC run in standby disabled
- ADC On demand disabled
- No sampling time compensation
- Disable the positive input sequence
- No reference compensation
- No gain/offset correction
- No added sampling time

Table 3-9. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to configuration struct to initialize to default values

3.6.4.2. Status Management

Function `adc_get_status()`

Retrieves the current module status.

```
uint32_t adc_get_status(
    struct adc_module *const module_inst)
```

Retrieves the status of the module, giving overall state information.

Table 3-10. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Returns

Bitmask of `ADC_STATUS_*` flags.

Table 3-11. Return Values

Return value	Description
<code>ADC_STATUS_RESULT_READY</code>	ADC result is ready to be read
<code>ADC_STATUS_WINDOW</code>	ADC has detected a value inside the set window range
<code>ADC_STATUS_OVERRUN</code>	ADC result has overrun

Function `adc_clear_status()`

Clears a module status flag.

```
void adc_clear_status(  
    struct adc_module *const module_inst,  
    const uint32_t status_flags)
```

Clears the given status flag of the module.

Table 3-12. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	status_flags	Bitmask of <code>ADC_STATUS_*</code> flags to clear

3.6.4.3. Enable, Disable, and Reset ADC Module, Start Conversion and Read Result

Function `adc_enable()`

Enables the ADC module.

```
enum status_code adc_enable(  
    struct adc_module *const module_inst)
```

Enables an ADC module that has previously been configured. If any internal reference is selected it will be enabled.

Table 3-13. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Function `adc_disable()`

Disables the ADC module.

```
enum status_code adc_disable(  
    struct adc_module *const module_inst)
```

Disables an ADC module that was previously enabled.

Table 3-14. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Function `adc_reset()`

Resets the ADC module.

```
enum status_code adc_reset(  
    struct adc_module *const module_inst)
```

Resets an ADC module, clearing all module state, and registers to their default values.

Table 3-15. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Function `adc_enable_events()`

Enables an ADC event input or output.

```
void adc_enable_events(
    struct adc_module *const module_inst,
    struct adc_events *const events)
```

Enables one or more input or output events to or from the ADC module. See [Struct `adc\_events`](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 3-16. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the ADC peripheral
[in]	events	Struct containing flags of events to enable

Function `adc_disable_events()`

Disables an ADC event input or output.

```
void adc_disable_events(
    struct adc_module *const module_inst,
    struct adc_events *const events)
```

Disables one or more input or output events to or from the ADC module. See [Struct `adc\_events`](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 3-17. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the ADC peripheral
[in]	events	Struct containing flags of events to disable

Function `adc_start_conversion()`

Starts an ADC conversion.

```
void adc_start_conversion(
    struct adc_module *const module_inst)
```

Starts a new ADC conversion.

Table 3-18. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Function adc_read()

Reads the ADC result.

```
enum status_code adc_read(
    struct adc_module *const module_inst,
    uint16_t * result)
```

Reads the result from an ADC conversion that was previously started.

Table 3-19. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[out]	result	Pointer to store the result value in

Returns

Status of the ADC read request.

Table 3-20. Return Values

Return value	Description
STATUS_OK	The result was retrieved successfully
STATUS_BUSY	A conversion result was not ready
STATUS_ERR_OVERFLOW	The result register has been overwritten by the ADC module before the result was read by the software

3.6.4.4. Runtime Changes of ADC Module**Function adc_flush()**

Flushes the ADC pipeline.

```
void adc_flush(
    struct adc_module *const module_inst)
```

Flushes the pipeline and restarts the ADC clock on the next peripheral clock edge. All conversions in progress will be lost. When flush is complete, the module will resume where it left off.

Table 3-21. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Function `adc_set_window_mode()`

Sets the ADC window mode.

```
void adc_set_window_mode(  
    struct adc_module *const module_inst,  
    const enum_adc_window_mode window_mode,  
    const int16_t window_lower_value,  
    const int16_t window_upper_value)
```

Sets the ADC window mode to a given mode and value range.

Table 3-22. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	window_mode	Window monitor mode to set
[in]	window_lower_value	Lower window monitor threshold value
[in]	window_upper_value	Upper window monitor threshold value

Function `adc_set_positive_input()`

Sets positive ADC input pin.

```
void adc_set_positive_input(  
    struct adc_module *const module_inst,  
    const enum_adc_positive_input positive_input)
```

Sets the positive ADC input pin selection.

Table 3-23. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	positive_input	Positive input pin

Function `adc_set_negative_input()`

Sets negative ADC input pin for differential mode.

```
void adc_set_negative_input(  
    struct adc_module *const module_inst,  
    const enum_adc_negative_input negative_input)
```

Sets the negative ADC input pin, when the ADC is configured in differential mode.

Table 3-24. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	negative_input	Negative input pin

3.6.4.5. Enable and Disable Interrupts

Function `adc_enable_interrupt()`

Enable interrupt.

```
void adc_enable_interrupt(  
    struct adc_module *const module_inst,  
    enum adc_interrupt_flag interrupt)
```

Enable the given interrupt request from the ADC module.

Table 3-25. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	interrupt	Interrupt to enable

Function `adc_disable_interrupt()`

Disable interrupt.

```
void adc_disable_interrupt(  
    struct adc_module *const module_inst,  
    enum adc_interrupt_flag interrupt)
```

Disable the given interrupt request from the ADC module.

Table 3-26. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	interrupt	Interrupt to disable

3.6.4.6. Callback Management

Function `adc_register_callback()`

Registers a callback.

```
void adc_register_callback(  
    struct adc_module *const module,  
    adc_callback_t callback_func,  
    enum adc_callback callback_type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled for the interrupt handler to call it when the condition for the callback is met.

Table 3-27. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

Function `adc_unregister_callback()`

Unregisters a callback.

```
void adc_unregister_callback(  
    struct adc_module * module,  
    enum adc_callback callback_type)
```

Unregisters a callback function which is implemented by the user.

Table 3-28. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_type	Callback type given by an enum

Function `adc_enable_callback()`

Enables callback.

```
void adc_enable_callback(  
    struct adc_module *const module,  
    enum adc_callback callback_type)
```

Enables the callback function registered by [adc_register_callback](#). The callback function will be called from the interrupt handler when the conditions for the callback type are met.

Table 3-29. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_type	Callback type given by an enum

Returns

Status of the operation.

Table 3-30. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID	If operation was not completed, due to invalid callback_type

Function `adc_disable_callback()`

Disables callback.

```
void adc_disable_callback(  
    struct adc_module *const module,  
    enum adc_callback callback_type)
```

Disables the callback function registered by the [adc_register_callback](#).

Table 3-31. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_type	Callback type given by an enum

Returns

Status of the operation.

Table 3-32. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID	If operation was not completed, due to invalid callback_type

3.6.4.7. Job Management**Function adc_read_buffer_job()**

Read multiple samples from ADC.

```
enum status_code adc_read_buffer_job(
    struct adc_module *const module_inst,
    uint16_t *buffer,
    uint16_t samples)
```

Read `samples` from the ADC into the `buffer`. If there is no hardware trigger defined (event action) the driver will retrigger the ADC conversion whenever a conversion is complete until `samples` has been acquired. To avoid jitter in the sampling frequency using an event trigger is advised.

Table 3-33. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	samples	Number of samples to acquire
[out]	buffer	Buffer to store the ADC samples

Returns

Status of the job start.

Table 3-34. Return Values

Return value	Description
STATUS_OK	The conversion job was started successfully and is in progress
STATUS_BUSY	The ADC is already busy with another job

Function `adc_get_job_status()`

Gets the status of a job.

```
enum status_code adc_get_job_status(  
    struct adc_module * module_inst,  
    enum adc_job_type type)
```

Gets the status of an ongoing or the last job.

Table 3-35. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	type	Type of to get status

Returns

Status of the job.

Function `adc_abort_job()`

Aborts an ongoing job.

```
void adc_abort_job(  
    struct adc_module * module_inst,  
    enum adc_job_type type)
```

Aborts an ongoing job.

Table 3-36. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	type	Type of job to abort

3.6.4.8. Positive Input Sequence

Function `adc_enable_positive_input_sequence()`

Enable positive input sequence mask for conversion.

```
void adc_enable_positive_input_sequence(  
    struct adc_module *const module_inst,  
    uint32_t positive_input_sequence_mask_enable)
```

The sequence start from the lowest input, and go to the next enabled input automatically when the conversion is done. If no bits are set the sequence is disabled.

Table 3-37. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	enable_seq_mask	Sequence mask

Function `adc_disable_positive_input_sequence()`

Disable positive input in the sequence.

```
void adc_disable_positive_input_sequence(  
    struct adc_module *const module_inst)
```

Disable positive input in the sequence.

Table 3-38. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Function `adc_get_sequence_status()`

Get ADC sequence status.

```
void adc_get_sequence_status(  
    struct adc_module *const module_inst,  
    bool * is_sequence_busy,  
    uint8_t *sequence_state)
```

Check if a sequence is done and get last conversion done in the sequence.

Table 3-39. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[out]	is_sequence_busy	Sequence busy status
[out]	sequence_state	This value identifies the last conversion done in the sequence

3.6.4.9. Function `adc_set_master_slave_mode()`

Set ADC master and slave mode.

```
void adc_set_master_slave_mode(  
    struct adc_module *const master_inst,  
    struct adc_module *const slave_inst,  
    enum adc_dual_mode_trigger_selection dualsel)
```

Enable ADC module Master-Slave Operation and select dual mode trigger.

Table 3-40. Parameters

Data direction	Parameter name	Description
[in]	master_inst	Pointer to the master ADC software instance struct
[in]	slave_inst	Pointer to the slave ADC software instance struct
[in]	dualsel	Dual mode trigger selection

3.6.5. Enumeration Definitions

3.6.5.1. Enum `adc_accumulate_samples`

Enum for the possible numbers of ADC samples to accumulate. This setting is only used when the [ADC_RESOLUTION_CUSTOM](#) resolution setting is used.

Table 3-41. Members

Enum value	Description
ADC_ACCUMULATE_DISABLE	No averaging
ADC_ACCUMULATE_SAMPLES_2	Average 2 samples
ADC_ACCUMULATE_SAMPLES_4	Average 4 samples
ADC_ACCUMULATE_SAMPLES_8	Average 8 samples
ADC_ACCUMULATE_SAMPLES_16	Average 16 samples
ADC_ACCUMULATE_SAMPLES_32	Average 32 samples
ADC_ACCUMULATE_SAMPLES_64	Average 64 samples
ADC_ACCUMULATE_SAMPLES_128	Average 128 samples
ADC_ACCUMULATE_SAMPLES_256	Average 256 samples
ADC_ACCUMULATE_SAMPLES_512	Average 512 samples
ADC_ACCUMULATE_SAMPLES_1024	Average 1024 samples

3.6.5.2. Enum `adc_callback`

Callback types for ADC callback driver.

Table 3-42. Members

Enum value	Description
ADC_CALLBACK_READ_BUFFER	Callback for buffer received
ADC_CALLBACK_WINDOW	Callback when window is hit
ADC_CALLBACK_ERROR	Callback for error

3.6.5.3. Enum `adc_clock_prescaler`

Enum for the possible clock prescaler values for the ADC.

Table 3-43. Members

Enum value	Description
ADC_CLOCK_PRESCALER_DIV2	ADC clock division factor 2
ADC_CLOCK_PRESCALER_DIV4	ADC clock division factor 4
ADC_CLOCK_PRESCALER_DIV8	ADC clock division factor 8

Enum value	Description
ADC_CLOCK_PRESCALER_DIV16	ADC clock division factor 16
ADC_CLOCK_PRESCALER_DIV32	ADC clock division factor 32
ADC_CLOCK_PRESCALER_DIV64	ADC clock division factor 64
ADC_CLOCK_PRESCALER_DIV128	ADC clock division factor 128
ADC_CLOCK_PRESCALER_DIV256	ADC clock division factor 256

3.6.5.4. Enum `adc_divide_result`

Enum for the possible division factors to use when accumulating multiple samples. To keep the same resolution for the averaged result and the actual input value, the division factor must be equal to the number of samples accumulated. This setting is only used when the [ADC_RESOLUTION_CUSTOM](#) resolution setting is used.

Table 3-44. Members

Enum value	Description
ADC_DIVIDE_RESULT_DISABLE	Don't divide result register after accumulation
ADC_DIVIDE_RESULT_2	Divide result register by 2 after accumulation
ADC_DIVIDE_RESULT_4	Divide result register by 4 after accumulation
ADC_DIVIDE_RESULT_8	Divide result register by 8 after accumulation
ADC_DIVIDE_RESULT_16	Divide result register by 16 after accumulation
ADC_DIVIDE_RESULT_32	Divide result register by 32 after accumulation
ADC_DIVIDE_RESULT_64	Divide result register by 64 after accumulation
ADC_DIVIDE_RESULT_128	Divide result register by 128 after accumulation

3.6.5.5. Enum `adc_dual_mode_trigger_selection`

Enum for the trigger selection in dual mode.

Table 3-45. Members

Enum value	Description
ADC_DUAL_MODE_BOTH	Start event or software trigger will start a conversion on both ADCs
ADC_DUAL_MODE_INTERLEAVE	START event or software trigger will alternately start a conversion on ADC0 and ADC1

3.6.5.6. Enum `adc_event_action`

Enum for the possible actions to take on an incoming event.

Table 3-46. Members

Enum value	Description
ADC_EVENT_ACTION_DISABLED	Event action disabled
ADC_EVENT_ACTION_FLUSH_START_CONV	Flush ADC and start conversion
ADC_EVENT_ACTION_START_CONV	Start conversion

3.6.5.7. Enum adc_interrupt_flag

Enum for the possible ADC interrupt flags.

Table 3-47. Members

Enum value	Description
ADC_INTERRUPT_RESULT_READY	ADC result ready
ADC_INTERRUPT_WINDOW	Window monitor match
ADC_INTERRUPT_OVERRUN	ADC result overwritten before read

3.6.5.8. Enum adc_job_type

Enum for the possible types of ADC asynchronous jobs that may be issued to the driver.

Table 3-48. Members

Enum value	Description
ADC_JOB_READ_BUFFER	Asynchronous ADC read into a user provided buffer

3.6.5.9. Enum adc_negative_input

Enum for the possible negative MUX input selections for the ADC.

Table 3-49. Members

Enum value	Description
ADC_NEGATIVE_INPUT_PIN0	ADC0 pin
ADC_NEGATIVE_INPUT_PIN1	ADC1 pin
ADC_NEGATIVE_INPUT_PIN2	ADC2 pin
ADC_NEGATIVE_INPUT_PIN3	ADC3 pin
ADC_NEGATIVE_INPUT_PIN4	ADC4 pin
ADC_NEGATIVE_INPUT_PIN5	ADC5 pin
ADC_NEGATIVE_INPUT_PIN6	ADC6 pin
ADC_NEGATIVE_INPUT_PIN7	ADC7 pin
ADC_NEGATIVE_INPUT_GND	Internal ground

3.6.5.10. Enum `adc_oversampling_and_decimation`

Enum for the possible numbers of bits resolution can be increased by when using oversampling and decimation.

Table 3-50. Members

Enum value	Description
<code>ADC_OVERSAMPLING_AND_DECIMATION_DISABLE</code>	Don't use oversampling and decimation mode
<code>ADC_OVERSAMPLING_AND_DECIMATION_1BIT</code>	1-bit resolution increase
<code>ADC_OVERSAMPLING_AND_DECIMATION_2BIT</code>	2-bit resolution increase
<code>ADC_OVERSAMPLING_AND_DECIMATION_3BIT</code>	3-bit resolution increase
<code>ADC_OVERSAMPLING_AND_DECIMATION_4BIT</code>	4-bit resolution increase

3.6.5.11. Enum `adc_positive_input`

Enum for the possible positive MUX input selections for the ADC.

Table 3-51. Members

Enum value	Description
<code>ADC_POSITIVE_INPUT_PIN0</code>	ADC0 pin
<code>ADC_POSITIVE_INPUT_PIN1</code>	ADC1 pin
<code>ADC_POSITIVE_INPUT_PIN2</code>	ADC2 pin
<code>ADC_POSITIVE_INPUT_PIN3</code>	ADC3 pin
<code>ADC_POSITIVE_INPUT_PIN4</code>	ADC4 pin
<code>ADC_POSITIVE_INPUT_PIN5</code>	ADC5 pin
<code>ADC_POSITIVE_INPUT_PIN6</code>	ADC6 pin
<code>ADC_POSITIVE_INPUT_PIN7</code>	ADC7 pin
<code>ADC_POSITIVE_INPUT_PIN8</code>	ADC8 pin
<code>ADC_POSITIVE_INPUT_PIN9</code>	ADC9 pin
<code>ADC_POSITIVE_INPUT_PIN10</code>	ADC10 pin
<code>ADC_POSITIVE_INPUT_PIN11</code>	ADC11 pin
<code>ADC_POSITIVE_INPUT_PIN12</code>	ADC12 pin
<code>ADC_POSITIVE_INPUT_PIN13</code>	ADC13 pin
<code>ADC_POSITIVE_INPUT_PIN14</code>	ADC14 pin
<code>ADC_POSITIVE_INPUT_PIN15</code>	ADC15 pin

Enum value	Description
ADC_POSITIVE_INPUT_PIN16	ADC16 pin
ADC_POSITIVE_INPUT_PIN17	ADC17 pin
ADC_POSITIVE_INPUT_PIN18	ADC18 pin
ADC_POSITIVE_INPUT_PIN19	ADC19 pin
ADC_POSITIVE_INPUT_PIN20	ADC20 pin.
ADC_POSITIVE_INPUT_PIN21	ADC21 pin
ADC_POSITIVE_INPUT_PIN22	ADC22 pin
ADC_POSITIVE_INPUT_PIN23	ADC23 pin
ADC_POSITIVE_INPUT_TEMP	Temperature reference.
ADC_POSITIVE_INPUT_BANDGAP	Bandgap voltage
ADC_POSITIVE_INPUT_SCALED COREVCC	1/4 scaled core supply
ADC_POSITIVE_INPUT_SCALED IOVCC	1/4 scaled I/O supply
ADC_POSITIVE_INPUT_DAC	DAC input

3.6.5.12. Enum `adc_reference`

Enum for the possible reference voltages for the ADC.

Table 3-52. Members

Enum value	Description
ADC_REFERENCE_INTREF	Internal Bandgap Reference
ADC_REFERENCE_INTVCC0	1/1.48V _{CC} reference
ADC_REFERENCE_INTVCC1	1/2V _{CC} (only for internal V _{CC} > 2.1V)
ADC_REFERENCE_AREFA	External reference A
ADC_REFERENCE_INTVCC2	VDDANA

3.6.5.13. Enum `adc_resolution`

Enum for the possible resolution values for the ADC.

Table 3-53. Members

Enum value	Description
ADC_RESOLUTION_12BIT	ADC 12-bit resolution
ADC_RESOLUTION_16BIT	ADC 16-bit resolution using oversampling and decimation
ADC_RESOLUTION_10BIT	ADC 10-bit resolution

Enum value	Description
ADC_RESOLUTION_8BIT	ADC 8-bit resolution
ADC_RESOLUTION_13BIT	ADC 13-bit resolution using oversampling and decimation
ADC_RESOLUTION_14BIT	ADC 14-bit resolution using oversampling and decimation
ADC_RESOLUTION_15BIT	ADC 15-bit resolution using oversampling and decimation
ADC_RESOLUTION_CUSTOM	ADC 16-bit result register for use with averaging. When using this mode the ADC result register will be set to 16-bit wide, and the number of samples to accumulate and the division factor is configured by the adc_config::accumulate_samples and adc_config::divide_result members in the configuration struct.

3.6.5.14. Enum `adc_window_mode`

Enum for the possible window monitor modes for the ADC.

Table 3-54. Members

Enum value	Description
ADC_WINDOW_MODE_DISABLE	No window mode
ADC_WINDOW_MODE_ABOVE_LOWER	RESULT > WINLT
ADC_WINDOW_MODE_BELOW_UPPER	RESULT < WINUT
ADC_WINDOW_MODE_BETWEEN	WINLT < RESULT < WINUT
ADC_WINDOW_MODE_BETWEEN_INVERTED	!(WINLT < RESULT < WINUT)

3.7. Extra Information for ADC Driver

3.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
LSB	Least Significant Bit
MSB	Most Significant Bit
DMA	Direct Memory Access

3.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

3.7.3. Errata

There are no errata related to this driver.

3.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

3.8. Examples for ADC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Analog-to-Digital Converter \(ADC\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for ADC - Basic](#)
- [Quick Start Guide for ADC - Callback](#)
- [Quick Start Guide for Using DMA with ADC/DAC](#)

3.8.1. Quick Start Guide for ADC - Basic

In this use case, the ADC will be configured with the following settings:

- 1V from internal bandgap reference
- Div 4 clock prescaler
- 12-bit resolution
- Window monitor disabled
- No gain
- Positive input on ADC PIN x (depend on default configuration)
- Negative input to GND (single ended)
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running disabled
- All events (input and generation) disabled
- Sleep operation disabled
- No reference compensation
- No gain/offset correction
- No added sampling time
- Pin scan mode disabled

3.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, outside of any functions:

```
struct adc_module adc_instance;
```

Copy-paste the following setup code to your user application:

```
void configure_adc(void)
{
    struct adc_config config_adc;
    adc_get_config_defaults(&config_adc);

    #if (SAMC21)
        adc_init(&adc_instance, ADC1, &config_adc);
    #else
        adc_init(&adc_instance, ADC, &config_adc);
    #endif

    adc_enable(&adc_instance);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_adc();
```

Workflow

1. Create a module software instance structure for the ADC module to store the ADC driver state while in use.

```
struct adc_module adc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the ADC module.
 1. Create an ADC module configuration struct, which can be filled out to adjust the configuration of a physical ADC peripheral.

```
struct adc_config config_adc;
```

2. Initialize the ADC configuration struct with the module's default values.

```
adc_get_config_defaults(&config_adc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set ADC configurations.

```
#if (SAMC21)
    adc_init(&adc_instance, ADC1, &config_adc);
#else
    adc_init(&adc_instance, ADC, &config_adc);
#endif
```

4. Enable the ADC module so that conversions can be made.

```
adc_enable(&adc_instance);
```

3.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
adc_start_conversion(&adc_instance);  
  
uint16_t result;  
  
do {  
    /* Wait for conversion to be done and read out result */  
} while (adc_read(&adc_instance, &result) == STATUS_BUSY);  
  
while (1) {  
    /* Infinite loop */  
}
```

Workflow

1. Start conversion.

```
adc_start_conversion(&adc_instance);
```

2. Wait until conversion is done and read result.

```
uint16_t result;  
  
do {  
    /* Wait for conversion to be done and read out result */  
} while (adc_read(&adc_instance, &result) == STATUS_BUSY);
```

3. Enter an infinite loop once the conversion is complete.

```
while (1) {  
    /* Infinite loop */  
}
```

3.8.2. Quick Start Guide for ADC - Callback

In this use case, the ADC will convert 128 samples using interrupt driven conversion. When all samples have been sampled, a callback will be called that signals the main application that conversion is complete.

The ADC will be set up as follows:

- $V_{CC}/2$ as reference
- Div 8 clock prescaler
- 12-bit resolution
- Window monitor disabled
- 1/2 gain
- Positive input on ADC PIN 0
- Negative input to GND (single ended)
- Averaging disabled
- Oversampling disabled
- Right adjust data

- Single-ended mode
- Free running disabled
- All events (input and generation) disabled
- Sleep operation disabled
- No reference compensation
- No gain/offset correction
- No added sampling time
- Pin scan mode disabled

3.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, outside of any functions:

```
struct adc_module adc_instance;

#define ADC_SAMPLES 128
uint16_t adc_result_buffer[ADC_SAMPLES];
```

Callback function:

```
volatile bool adc_read_done = false;

void adc_complete_callback(
    struct adc_module *const module)
{
    adc_read_done = true;
}
```

Copy-paste the following setup code to your user application:

```
void configure_adc(void)
{
    struct adc_config config_adc;
    adc_get_config_defaults(&config_adc);

    #if (!SAML21) && (!SAML22) && (!SAMC21) && (!SAMR30)
        config_adc.gain_factor = ADC_GAIN_FACTOR_DIV2;
    #endif
    config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV8;
    config_adc.reference = ADC_REFERENCE_INTVCC1;
    #if (SAMC21)
        config_adc.positive_input = ADC_POSITIVE_INPUT_PIN5;
    #else
        config_adc.positive_input = ADC_POSITIVE_INPUT_PIN6;
    #endif
    config_adc.resolution = ADC_RESOLUTION_12BIT;

    #if (SAMC21)
        adc_init(&adc_instance, ADC1, &config_adc);
    #else
        adc_init(&adc_instance, ADC, &config_adc);
    #endif

    adc_enable(&adc_instance);
}
```

```
void configure_adc_callbacks(void)
{
    adc_register_callback(&adc_instance,
        adc_complete_callback, ADC_CALLBACK_READ_BUFFER);
    adc_enable_callback(&adc_instance, ADC_CALLBACK_READ_BUFFER);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_adc();
configure_adc_callbacks();
```

Workflow

1. Create a module software instance structure for the ADC module to store the ADC driver state while in use.

```
struct adc_module adc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a buffer for the ADC samples to be stored in by the driver asynchronously.

```
#define ADC_SAMPLES 128
uint16_t adc_result_buffer[ADC_SAMPLES];
```

3. Create a callback function that will be called each time the ADC completes an asynchronous read job.

```
volatile bool adc_read_done = false;

void adc_complete_callback(
    struct adc_module *const module)
{
    adc_read_done = true;
}
```

4. Configure the ADC module.

1. Create an ADC module configuration struct, which can be filled out to adjust the configuration of a physical ADC peripheral.

```
struct adc_config config_adc;
```

2. Initialize the ADC configuration struct with the module's default values.

```
adc_get_config_defaults(&config_adc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Change the ADC module configuration to suit the application.

```
#if (!SAML21) && (!SAML22) && (!SAMC21) && (!SAMR30)
    config_adc.gain_factor = ADC_GAIN_FACTOR_DIV2;
#endif
config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV8;
config_adc.reference = ADC_REFERENCE_INTVCC1;
#if (SAMC21)
    config_adc.positive_input = ADC_POSITIVE_INPUT_PIN5;
#else
    config_adc.positive_input = ADC_POSITIVE_INPUT_PIN6;
```

```
#endif
config_adc.resolution      = ADC_RESOLUTION_12BIT;
```

4. Set ADC configurations.

```
#if (SAMC21)
    adc_init(&adc_instance, ADC1, &config_adc);
#else
    adc_init(&adc_instance, ADC, &config_adc);
#endif
```

5. Enable the ADC module so that conversions can be made.

```
adc_enable(&adc_instance);
```

5. Register and enable the ADC Read Buffer Complete callback handler.

1. Register the user-provided Read Buffer Complete callback function with the driver, so that it will be run when an asynchronous buffer read job completes.

```
adc_register_callback(&adc_instance,
    adc_complete_callback, ADC_CALLBACK_READ_BUFFER);
```

2. Enable the Read Buffer Complete callback so that it will generate callbacks.

```
adc_enable_callback(&adc_instance, ADC_CALLBACK_READ_BUFFER);
```

3.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();
adc_read_buffer_job(&adc_instance, adc_result_buffer, ADC_SAMPLES);

while (adc_read_done == false) {
    /* Wait for asynchronous ADC read to complete */
}

while (1) {
    /* Infinite loop */
}
```

Workflow

1. Enable global interrupts, so that callbacks can be generated by the driver.

```
system_interrupt_enable_global();
```

2. Start an asynchronous ADC conversion, to store ADC samples into the global buffer and generate a callback when complete.

```
adc_read_buffer_job(&adc_instance, adc_result_buffer, ADC_SAMPLES);
```

3. Wait until the asynchronous conversion is complete.

```
while (adc_read_done == false) {
    /* Wait for asynchronous ADC read to complete */
}
```

4. Enter an infinite loop once the conversion is complete.

```
while (1) {
    /* Infinite loop */
}
```

3.8.3. Quick Start Guide for Using DMA with ADC/DAC

The supported board list:

- SAM D21 Xplained Pro
- SAM D11 Xplained Pro
- SAM L21 Xplained Pro
- SAM DA1 Xplained Pro
- SAM C21 Xplained Pro

This quick start will convert an analog input signal from AIN4 and output the converted value to DAC on PA2. The data between ADC and DAC will be transferred through DMA instead of a CPU intervene.

The ADC will be configured with the following settings:

- 1/2 VDDANA
- Div 16 clock prescaler
- 10-bit resolution
- Window monitor disabled
- No gain
- Positive input on ADC AIN4
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running enable
- All events (input and generation) disabled
- Sleep operation disabled
- No reference compensation
- No gain/offset correction
- No added sampling time
- Pin scan mode disabled

The DAC will be configured with the following settings:

- Analog V_{CC} as reference
- Internal output disabled
- Drive the DAC output to PA2
- Right adjust data
- The output buffer is disabled when the chip enters STANDBY sleep mode

The DMA will be configured with the following settings:

- Move data from peripheral to peripheral
- Using ADC result ready trigger
- Using DMA priority level 0
- Beat transfer will be triggered on each trigger
- Loopback descriptor for DAC conversion

3.8.3.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, outside of any functions:

```
struct dac_module dac_instance;
```

```
struct adc_module adc_instance;
```

```
struct dma_resource example_resource;
```

```
COMPILER_ALIGNED(16)  
DmacDescriptor example_descriptor SECTION_DMAC_DESCRIPTOR;
```

Copy-paste the following setup code to your user application:

```
void configure_adc(void)  
{  
    struct adc_config config_adc;  
  
    adc_get_config_defaults(&config_adc);  
  
#if !(SAML21)  
#if !(SAMC21)  
    config_adc.gain_factor      = ADC_GAIN_FACTOR_DIV2;  
#endif  
    config_adc.resolution      = ADC_RESOLUTION_10BIT;  
#endif  
    config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV16;  
    config_adc.reference       = ADC_REFERENCE_INTVCC1;  
    config_adc.positive_input  = ADC_POSITIVE_INPUT_PIN4;  
    config_adc.freerunning     = true;  
    config_adc.left_adjust     = false;  
  
#if (SAMC21)  
    adc_init(&adc_instance, ADC1, &config_adc);  
#else  
    adc_init(&adc_instance, ADC, &config_adc);  
#endif  
  
    adc_enable(&adc_instance);  
}  
  
void configure_dac(void)  
{  
    struct dac_config config_dac;  
  
    dac_get_config_defaults(&config_dac);  
  
#if (SAML21)  
    config_dac.reference = DAC_REFERENCE_INTREF;  
#else  
    config_dac.reference = DAC_REFERENCE_AVCC;  
#endif  
  
    dac_init(&dac_instance, DAC, &config_dac);  
}  
  
void configure_dac_channel(void)  
{  
    struct dac_chan_config config_dac_chan;  
  
    dac_chan_get_config_defaults(&config_dac_chan);
```



```

        dac_chan_set_config(&dac_instance, DAC_CHANNEL_0, &config_dac_chan);
        dac_chan_enable(&dac_instance, DAC_CHANNEL_0);
    }

void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    #if (SAMC21)
        config.peripheral_trigger = ADC1_DMACH_ID_RESRDY;
    #else
        config.peripheral_trigger = ADC_DMACH_ID_RESRDY;
    #endif
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(resource, &config);
}

void setup_transfer_descriptor(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.src_increment_enable = false;
    descriptor_config.block_transfer_count = 1000;
    descriptor_config.source_address = (uint32_t)(&adc_instance.hw-
>RESULT.reg);
    #if (SAML21)
        descriptor_config.destination_address = (uint32_t)(&dac_instance.hw-
>DATA[DAC_CHANNEL_0].reg);
    #else
        descriptor_config.destination_address = (uint32_t)(&dac_instance.hw-
>DATA.reg);
    #endif
    descriptor_config.next_descriptor_address = (uint32_t)descriptor;

    dma_descriptor_create(descriptor, &descriptor_config);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_adc();
configure_dac();
configure_dac_channel();
dac_enable(&dac_instance);
configure_dma_resource(&example_resource);
setup_transfer_descriptor(&example_descriptor);
dma_add_descriptor(&example_resource, &example_descriptor);

```

Workflow

Configure the ADC

1. Create a module software instance structure for the ADC module to store the ADC driver state while it is in use.

```
struct adc_module adc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the ADC module.
 1. Create an ADC module configuration struct, which can be filled out to adjust the configuration of a physical ADC peripheral.

```
struct adc_config config_adc;
```

2. Initialize the ADC configuration struct with the module's default values.

```
adc_get_config_defaults(&config_adc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra configurations.

```
#if !(SAML21)
#if !(SAMC21)
    config_adc.gain_factor      = ADC_GAIN_FACTOR_DIV2;
#endif
    config_adc.resolution      = ADC_RESOLUTION_10BIT;
#endif
    config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV16;
    config_adc.reference       = ADC_REFERENCE_INTVCC1;
    config_adc.positive_input   = ADC_POSITIVE_INPUT_PIN4;
    config_adc.freerunning     = true;
    config_adc.left_adjust      = false;
```

4. Set ADC configurations.

```
#if (SAMC21)
    adc_init(&adc_instance, ADC1, &config_adc);
#else
    adc_init(&adc_instance, ADC, &config_adc);
#endif
```

5. Enable the ADC module so that conversions can be made.

```
adc_enable(&adc_instance);
```

Configure the DAC

1. Create a module software instance structure for the DAC module to store the DAC driver state while it is in use.

```
struct dac_module dac_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the DAC module.

1. Create a DAC module configuration struct, which can be filled out to adjust the configuration of a physical DAC peripheral.

```
struct dac_config config_dac;
```

2. Initialize the DAC configuration struct with the module's default values.

```
dac_get_config_defaults(&config_dac);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra DAC configurations.

```
#if (SAML21)
    config_dac.reference = DAC_REFERENCE_INTREF;
#else
    config_dac.reference = DAC_REFERENCE_AVCC;
#endif
```

4. Set DAC configurations to DAC instance.

```
dac_init(&dac_instance, DAC, &config_dac);
```

5. Enable the DAC module so that channels can be configured.

```
dac_enable(&dac_instance);
```

3. Configure the DAC channel.

1. Create a DAC channel configuration struct, which can be filled out to adjust the configuration of a physical DAC output channel.

```
struct dac_chan_config config_dac_chan;
```

2. Initialize the DAC channel configuration struct with the module's default values.

```
dac_chan_get_config_defaults(&config_dac_chan);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Configure the DAC channel with the desired channel settings.

```
dac_chan_set_config(&dac_instance, DAC_CHANNEL_0,
&config_dac_chan);
```

4. Enable the DAC channel so that it can output a voltage.

```
dac_chan_enable(&dac_instance, DAC_CHANNEL_0);
```

Configure the DMA

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

2. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra configurations for the DMA resource. ADC_DMAC_ID_RESRDY trigger causes a beat transfer in this example.

```
#if (SAMC21)
    config.peripheral_trigger = ADC1_DMAC_ID_RESRDY;
#else
    config.peripheral_trigger = ADC_DMAC_ID_RESRDY;
#endif
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

4. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

5. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

6. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

7. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.src_increment_enable = false;
descriptor_config.block_transfer_count = 1000;
descriptor_config.source_address = (uint32_t)(&adc_instance.hw-
>RESULT.reg);
#if (SAML21)
    descriptor_config.destination_address = (uint32_t)
(&dac_instance.hw->DATA[DAC_CHANNEL_0].reg);
#else
    descriptor_config.destination_address = (uint32_t)
(&dac_instance.hw->DATA.reg);
#endif
descriptor_config.next_descriptor_address = (uint32_t)descriptor;
```

8. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

9. Add DMA descriptor to DMA resource.

```
dma_add_descriptor(&example_resource, &example_descriptor);
```

3.8.3.2. Use Case

Code

Copy-paste the following code to your user application:

```
adc_start_conversion(&adc_instance);
dma_start_transfer_job(&example_resource);

while (true) {
}
```

Workflow

1. Start ADC conversion.

```
adc_start_conversion(&adc_instance);
```

2. Start the transfer job.

```
dma_start_transfer_job(&example_resource);
```

3. Enter endless loop.

```
while (true) {  
}
```

4. SAM Advanced Encryption Standard (AES) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Advanced Encryption Standard functionality. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The Advanced Encryption Standard module supports all five confidentiality modes of operation for symmetrical key block cipher algorithms (as specified in the NIST Special Publication 800-38A Recommendation):

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Output Feedback (OFB)
- Cipher Feedback (CFB)
- Counter (CTR)

The following peripheral is used by this module:

- AES (Advanced Encryption Standard)

The following devices can use this module:

- Atmel | SMART SAM L21
- Atmel | SMART SAM L22

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

4.1. Prerequisites

There are no prerequisites for this module.

4.2. Module Overview

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It is compliant with the American FIPS (Federal Information Processing Standard) Publication 197 specification.

The AES supports all five confidentiality modes of operation for symmetrical key block cipher algorithms (as specified in the NIST Special Publication 800-38A Recommendation):

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Output Feedback (OFB)

- Cipher Feedback (CFB)
- Counter (CTR)

Data transfers both to and from the AES module can occur using the peripheral DMA controller channels, thus minimizing processor intervention for large data buffer transfers.

As soon as the initialization vector, the input data and the key are configured, the encryption/decryption process may be started. Once the process has completed the encrypted/decrypted data can be read out via registers or through DMA channels.

4.2.1. Encryption and Decryption

The AES is capable of using cryptographic keys of 128/192/256 bits to encrypt and decrypt data in blocks of 128 bits. In Cipher Feedback Mode (CFB), five data sizes are possible (8, 16, 32, 64, or 128 bits).

The input to the encryption processes of the CBC, CFB, and OFB modes includes, in addition to the plaintext, a 128-bit data block called the Initialization Vector (IV). The Initialization Vector is used in the initial step in the encryption of a message and in the corresponding decryption of the message.

There are three encryption/decryption start modes:

- Manual Mode: Start encryption/decryption manually
- Auto Start Mode: Once the correct number of input data registers is written, processing is automatically started, DMA operation uses this mode
- Last Output Data Mode (LOD): This mode is used to generate message authentication code (MAC) on data in CCM mode of operation

4.2.2. Hardware Countermeasures

The AES module features four types of hardware countermeasures that are useful for protecting data against differential power analysis attacks:

- Type 1: Randomly add one cycle to data processing
- Type 2: Randomly add one cycle to data processing (other version)
- Type 3: Add a random number of clock cycles to data processing, subject to a maximum of 11/13/15 clock cycles for key sizes of 128/192/256 bits
- Type 4: Add random spurious power consumption during data processing

4.2.3. Galois Counter Mode (GCM)

GCM is comprised of the AES engine in CTR mode along with a universal hash function (GHASH engine) that is defined over a binary Galois field to produce a message authentication tag. The GHASH engine processes data packets after the AES operation. GCM provides assurance of the confidentiality of data through the AES Counter mode of operation for DRAFT 920 encryption. Authenticity of the confidential data is assured through the GHASH engine. Refer to the NIST Special Publication 800-38D Recommendation for more complete information.

4.3. Special Considerations

There are no special considerations for this module.

4.4. Extra Information

For extra information, see [Extra Information for Advanced Encryption Standard](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

4.5. Examples

For a list of examples related to this driver, see [Examples for Advanced Encryption Standard](#).

4.6. API Overview

4.6.1. Variable and Type Definitions

4.6.1.1. Type `aes_callback_t`

```
typedef void(* aes_callback_t )(void)
```

AES interrupt callback function type.

4.6.2. Structure Definitions

4.6.2.1. Struct `aes_config`

AES Configuration structure.

Table 4-1. Members

Type	Name	Description
enum aes_cfb_size	<code>cfb_size</code>	Cipher feedback data size
enum aes_countermeasure_type	<code>ctype</code>	Countermeasure type
bool	<code>enable_key_gen</code>	Enable key generation
bool	<code>enable_xor_key</code>	Enable XOR key
enum aes_encrypt_mode	<code>encrypt_mode</code>	AES data mode (decryption or encryption)
enum aes_key_size	<code>key_size</code>	AES key size
bool	<code>lod</code>	Last output data mode enable/disable
enum aes_operation_mode	<code>opmode</code>	AES cipher operation mode
enum aes_start_mode	<code>start_mode</code>	Start mode

4.6.3. Macro Definitions

4.6.3.1. Module Status Flags

AES status flags, returned by [aes_get_status\(\)](#) and cleared by [aes_clear_status\(\)](#).

Macro AES_ENCRYPTION_COMPLETE

```
#define AES_ENCRYPTION_COMPLETE
```

AES encryption complete.

Macro AES_GF_MULTI_COMPLETE

```
#define AES_GF_MULTI_COMPLETE
```

AES GF multiplication complete.

4.6.4. Function Definitions

4.6.4.1. Configuration and Initialization

Function aes_get_config_defaults()

Initializes an AES configuration structure to defaults.

```
void aes_get_config_defaults(  
    struct aes_config *const config)
```

Initializes the specified AES configuration structure to a set of known default values.

Note: This function should be called to initialize **all** new instances of AES configuration structures before they are further modified by the user application.

The default configuration is as follows:

- Data encryption
- 128-bit AES key size
- 128-bit cipher feedback size
- Manual start mode
- Electronic Codebook (ECB) mode
- All countermeasures are enabled
- XRO key is disabled
- Key generation is disabled
- Last output data mode is disabled

Table 4-2. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to an AES configuration structure

Function aes_set_config()

Configure the AES module.

```
void aes_set_config(  
    struct aes_module *const module,  
    Aes *const hw,  
    struct aes_config *const config)
```

Table 4-3. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Module hardware register base address pointer
[in]	config	Pointer to an AES configuration structure

Function aes_init()

Initialize the AES module.

```
void aes_init(
    struct aes_module *const module,
    Aes *const hw,
    struct aes_config *const config)
```

Table 4-4. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Module hardware register base address pointer
[in]	config	Pointer to an AES configuration structure

4.6.4.2. Start, Enable, and Write**Function aes_start()**

Start a manual encryption/decryption process.

```
void aes_start(
    struct aes_module *const module)
```

Table 4-5. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Function aes_set_new_message()

Notifies the module that the next input data block is the beginning of a new message.

```
void aes_set_new_message(
    struct aes_module *const module)
```

Table 4-6. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Function `aes_clear_new_message()`

Clear the indication of the beginning for a new message.

```
void aes_clear_new_message(  
    struct aes_module *const module)
```

Table 4-7. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Function `aes_enable()`

Enable the AES module.

```
void aes_enable(  
    struct aes_module *const module)
```

Table 4-8. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `aes_disable()`

Disable the AES module.

```
void aes_disable(  
    struct aes_module *const module)
```

Table 4-9. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Function `aes_write_key()`

Write the 128/192/256-bit cryptographic key.

```
void aes_write_key(  
    struct aes_module *const module,  
    const uint32_t * key)
```

Table 4-10. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	key	Pointer to 4/6/8 contiguous 32-bit words

Note: The key size depends on the current AES configuration.

Function aes_write_init_vector()

Write the initialization vector (for the CBC, CFB, OFB, CTR, and GCM cipher modes).

```
void aes_write_init_vector(  
    struct aes_module *const module,  
    const uint32_t * vector)
```

Table 4-11. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	vector	Pointer to four contiguous 32-bit words

Function aes_write_input_data()

Write the input data (four consecutive 32-bit words).

```
void aes_write_input_data(  
    struct aes_module *const module,  
    const uint32_t * p_input_data_buffer)
```

Table 4-12. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	input_data_buffer	Pointer to an input data buffer

Function aes_read_output_data()

Read the output data.

```
void aes_read_output_data(  
    struct aes_module *const module,  
    uint32_t * p_output_data_buffer)
```

Note: The data buffer that holds the processed data must be large enough to hold four consecutive 32-bit words.

Table 4-13. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	output_data_buffer	Pointer to an output buffer

Function aes_write_random_seed()

Write AES random seed.

```
void aes_write_random_seed(  
    struct aes_module *const module,  
    uint32_t seed)
```

Table 4-14. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	seed	Seed for the random number generator

4.6.4.3. Status Management

Function aes_get_status()

Retrieves the current module status.

```
uint32_t aes_get_status(
    struct aes_module *const module)
```

Retrieves the status of the module, giving overall state information.

Table 4-15. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Table 4-16. Return Values

Return value	Description
AES_ENCRYPTION_COMPLETE	AES encryption complete
AES_GF_MULTI_COMPLETE	AES GF multiplication complete

Function aes_clear_status()

Clears a module status flag.

```
void aes_clear_status(
    struct aes_module *const module,
    const uint32_t status_flags)
```

Clears the given status flag of the module.

Table 4-17. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	status_flags	Bitmask flags to clear

4.6.4.4. Galois Counter Mode

Function aes_gcm_read_ghash()

Get the AES GCM Hash Value.

```
uint32_t aes_gcm_read_ghash(
    struct aes_module *const module,
    uint32_t id)
```

Table 4-18. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the GHASH array (range 0 to 3)

Returns

The content of the GHASHRx[x = 0...3] value.

Function aes_gcm_write_ghash()

Set the AES GCM Hash Value.

```
void aes_gcm_write_ghash(
    struct aes_module *const module,
    uint32_t id,
    uint32_t ghash)
```

Table 4-19. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the GHASHx array (range 0 to 3)
[in]	ghash	GCM hash value

Function aes_gcm_read_hash_key()

Get AES GCM Hash key.

```
uint32_t aes_gcm_read_hash_key(
    struct aes_module *const module,
    uint32_t id)
```

Table 4-20. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the Hash key array (range 0 to 3)

Returns

The contents of the HASHKEYx[x = 0...3] specified.

Function aes_gcm_write_hash_key()

Set the AES GCM Hash key.

```
void aes_gcm_write_hash_key(
    struct aes_module *const module,
    uint32_t id,
    uint32_t key)
```

Table 4-21. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the Hash key array (range 0 to 3)
[in]	key	GCM Hash key

Function aes_gcm_read_cipher_len()

Get the AES GCM cipher length.

```
uint32_t aes_gcm_read_cipher_len(
    struct aes_module *const module)
```

Table 4-22. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Returns

The contents of the HASHKEYx[x = 0...3] specified.

Function aes_gcm_write_cipher_len()

Set the AES GCM cipher length.

```
void aes_gcm_write_cipher_len(
    struct aes_module *const module,
    uint32_t len)
```

Table 4-23. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	len	Cipher length

Function aes_gcm_set_end_message_status()

Set GCM end of input message status.

```
void aes_gcm_set_end_message_status(
    struct aes_module *const module)
```

Table 4-24. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Function `aes_gcm_clear_end_message_status()`

Clear GCM end of input message status.

```
void aes_gcm_clear_end_message_status(  
    struct aes_module *const module)
```

Table 4-25. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Function `aes_gcm_set_gf_multiplication()`

Set GF multiplication of GCM mode.

```
void aes_gcm_set_gf_multiplication(  
    struct aes_module *const module)
```

Table 4-26. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

4.6.4.5. Callback Configuration and Initialization

Function `aes_register_callback()`

```
enum status_code aes_register_callback(  
    const aes_callback_t callback,  
    const enum aes_callback_type type)
```

Function `aes_unregister_callback()`

```
enum status_code aes_unregister_callback(  
    const aes_callback_t callback,  
    const enum aes_callback_type type)
```

4.6.4.6. Callback Enabling and Disabling

Function `aes_enable_callback()`

Enable an AES callback.

```
enum status_code aes_enable_callback(  
    struct aes_module *const module,  
    const enum aes_callback_type type)
```

Table 4-27. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	type	Callback source type

Returns

Status of the callback enable operation.

Table 4-28. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function aes_disable_callback()

Disable an AES callback.

```
enum status_code aes_disable_callback(
    struct aes_module *const module,
    const enum aes_callback_type type)
```

Table 4-29. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	type	Callback source type

Returns

Status of the callback enable operation.

Table 4-30. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

4.6.5. Enumeration Definitions**4.6.5.1. Enum aes_callback_type**

AES callback type.

Table 4-31. Members

Enum value	Description
AES_CALLBACK_ENCRYPTION_COMPLETE	Encryption complete callback
AES_CALLBACK_GF_MULTI_COMPLETE	GF Multiplication Complete callback

4.6.5.2. Enum aes_cfb_size

AES Cipher FeedBack (CFB) size.

Table 4-32. Members

Enum value	Description
AES_CFB_SIZE_128	Cipher feedback data size is 128-bit
AES_CFB_SIZE_64	Cipher feedback data size is 64-bit
AES_CFB_SIZE_32	Cipher feedback data size is 32-bit
AES_CFB_SIZE_16	Cipher feedback data size is 16-bit
AES_CFB_SIZE_8	Cipher feedback data size is 8-bit

4.6.5.3. Enum aes_countermeasure_type

AES countermeasure type

Table 4-33. Members

Enum value	Description
AES_COUNTERMEASURE_TYPE_disabled	Countermeasure type all disabled
AES_COUNTERMEASURE_TYPE_1	Countermeasure1 enabled
AES_COUNTERMEASURE_TYPE_2	Countermeasure2 enabled
AES_COUNTERMEASURE_TYPE_3	Countermeasure3 enabled
AES_COUNTERMEASURE_TYPE_4	Countermeasure4 enabled
AES_COUNTERMEASURE_TYPE_ALL	Countermeasure type all enabled

4.6.5.4. Enum aes_encrypt_mode

AES processing mode.

Table 4-34. Members

Enum value	Description
AES_DECRYPTION	Decryption of data will be performed
AES_ENCRYPTION	Encryption of data will be performed

4.6.5.5. Enum aes_key_size

AES cryptographic key size.

Table 4-35. Members

Enum value	Description
AES_KEY_SIZE_128	AES key size is 128-bit
AES_KEY_SIZE_192	AES key size is 192-bit
AES_KEY_SIZE_256	AES key size is 256-bit

4.6.5.6. Enum aes_operation_mode

AES operation mode.

Table 4-36. Members

Enum value	Description
AES_ECB_MODE	Electronic Codebook (ECB)
AES_CBC_MODE	Cipher Block Chaining (CBC)
AES_OFB_MODE	Output Feedback (OFB)
AES_CFB_MODE	Cipher Feedback (CFB)
AES_CTR_MODE	Counter (CTR)
AES_CCM_MODE	Counter (CCM)
AES_GCM_MODE	Galois Counter Mode (GCM)

4.6.5.7. Enum aes_start_mode

AES start mode.

Table 4-37. Members

Enum value	Description
AES_MANUAL_START	Manual start mode
AES_AUTO_START	Auto start mode

4.7. Extra Information for Advanced Encryption Standard

4.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Definition
AAD	Additional Authenticated Data
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CTR	Counter
DMA	Direct Memory Access
DMAC	DMA Controller
ECB	Electronic Codebook
GCM	Galois Counter Mode

Acronym	Definition
OFB	Output Feedback
QSG	Quick Start Guide

4.7.2. Dependencies

This driver has the following dependencies:

- None

4.7.3. Errata

There are no errata related to this driver.

4.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial release

4.8. Examples for Advanced Encryption Standard

This is a list of the available Quick Start Guides (QSGs) and example applications for [SAM Advanced Encryption Standard \(AES\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for AES - Basic](#)
- [Quick Start Guide for AES - Callback](#)
- [Quick Start Guide for AES - DMA](#)

4.8.1. Quick Start Guide for AES - Basic

The supported board list:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro

This example demonstrates how to use the AES driver to perform:

- ECB encryption and decryption

Upon startup, the program uses the USART driver to display application output message from which ECB encryption/decryption modes can be tested.

4.8.1.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, outside of any functions:

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};
/* @} */
```

Add to the main application source file, outside of any functions:

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];

/* State indicate */
volatile bool state = false;
/* AES configuration */
struct aes_config g_aes_cfg;
/* AES instance*/
struct aes_module aes_instance;
struct usart_module usart_instance;
```

Copy-paste the following setup code to your user application:

```
static void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}

static void ecb_mode_test(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB cipher mode\r\n");
}
```

```

printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance,AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */

aes_set_new_message(&aes_instance);
/* Write the data to be ciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_plain_text);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the encryption process. */
while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
}
aes_read_output_data(&aes_instance,output_data);

if ((ref_cipher_text_ecb[0] != output_data[0]) ||
    (ref_cipher_text_ecb[1] != output_data[1]) ||
    (ref_cipher_text_ecb[2] != output_data[2]) ||
    (ref_cipher_text_ecb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
printf("\r\n-----\r\n");
printf("- 128bit cryptographic key\r\n");
printf("- ECB decipher mode\r\n");
printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance,AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */

/* Write the data to be deciphered to the input data registers. */

```

```

aes_write_input_data(&aes_instance, ref_cipher_text_ecb);

/* Wait for the end of the decryption process. */
while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
}
aes_read_output_data(&aes_instance, output_data);

/* check the result. */
if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||
    (ref_plain_text[2] != output_data[2]) ||
    (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
}

```

Add to user application initialization (typically the start of `main()`):

```

/* Initialize the system and console*/
system_init();
configure_usart();

aes_get_config_defaults(&g_aes_cfg);

aes_init(&aes_instance, AES, &g_aes_cfg);
aes_enable(&aes_instance);

```

Workflow

1. Define sample data from NIST-800-38A appendix F for ECB mode.

```

#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};
/* @} */

```

2. Create related module variable and software instance structure.

```

/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];

```

```

/* State indicate */
volatile bool state = false;
/* AES configuration */
struct aes_config g_aes_cfg;
/* AES instance*/
struct aes_module aes_instance;
struct usart_module usart_instance;

```

3. Configure, initialize, and enable AES module.

1. Configuration AES struct, which can be filled out to adjust the configuration of a physical AES peripheral.

```
aes_get_config_defaults(&g_aes_cfg);
```

2. Initialize the AES configuration struct with the module's default values.

```
aes_init(&aes_instance, AES, &g_aes_cfg);
```

3. Enable the AES module.

```
aes_enable(&aes_instance);
```

4.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```

/* Run ECB mode test*/
ecb_mode_test();

```

Workflow

1. Configure ECB mode encryption and run test.

```

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */

aes_set_new_message(&aes_instance);
/* Write the data to be ciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_plain_text);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the encryption process. */
while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
}
aes_read_output_data(&aes_instance, output_data);

if ((ref_cipher_text_ecb[0] != output_data[0]) ||
    (ref_cipher_text_ecb[1] != output_data[1]) ||

```



```

        (ref_cipher_text_ecb[2] != output_data[2]) ||
        (ref_cipher_text_ecb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

```

2. Configure ECB mode decryption and run test.

```

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */

/* Write the data to be deciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_cipher_text_ecb);

/* Wait for the end of the decryption process. */
while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
}
aes_read_output_data(&aes_instance, output_data);

/* check the result. */
if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||
    (ref_plain_text[2] != output_data[2]) ||
    (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

```

4.8.2. Quick Start Guide for AES - Callback

The supported board list:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro

This example demonstrates how to use the AES driver to perform:

- ECB encryption and decryption
- CBC encryption and decryption
- CFB128 encryption and decryption
- OFB encryption and decryption
- CTR encryption and decryption

Upon startup, the program uses the USART driver to display application output message from which several encryption/decryption modes can be tested.

4.8.2.1. Quick Start Callback

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, outside of any functions:

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

uint32_t ref_cipher_text_cbc[AES_EXAMPLE_REFBUF_SIZE] = {
    0xacab4976,
    0x46b21981,
    0x9b8ee9ce,
    0x7d19e912
};

uint32_t ref_cipher_text_cfb128[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ofb[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ctr[AES_EXAMPLE_REFBUF_SIZE] = {
    0x91614d87,
    0x26e320b6,
    0x6468ef1b,
    0xceb60d99
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};

const uint32_t init_vector[4] = {
    0x03020100,
```

```

    0x07060504,
    0x0b0a0908,
    0x0f0e0d0c
};

const uint32_t init_vector_ctr[4] = {
    0xf3f2f1f0,
    0xf7f6f5f4,
    0xfbfaf9f8,
    0xffffefdfc
};

/* @} */

```

Add to the main application source file, outside of any functions:

```

/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;

struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;

```

Copy-paste the following setup code to your user application:

```

static void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}

static void aes_callback(void)
{
    /* Read the output. */
    aes_read_output_data(&aes_instance, output_data);
    state = true;
}

static void ecb_mode_test(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----\r\n");

    state = false;

    /* Configure the AES. */
}

```

```

g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);
/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */

aes_set_new_message(&aes_instance);
/* Write the data to be ciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_plain_text);
aes_clear_new_message(&aes_instance);

/* Wait for the end of the encryption process. */
while (false == state) {
}

if ((ref_cipher_text_ecb[0] != output_data[0]) ||
    (ref_cipher_text_ecb[1] != output_data[1]) ||
    (ref_cipher_text_ecb[2] != output_data[2]) ||
    (ref_cipher_text_ecb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

printf("\r\n-----\r\n");
printf("- 128bit cryptographic key\r\n");
printf("- ECB decipher mode\r\n");
printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */
aes_set_new_message(&aes_instance);
/* Write the data to be deciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_cipher_text_ecb);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the decryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||

```

```

        (ref_plain_text[2] != output_data[2]) ||
        (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
}

static void cbc_mode_test(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CBC cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_MANUAL_START;
    g_aes_cfg.opmode = AES_CBC_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance, AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector);

    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_set_new_message(&aes_instance);
    aes_start(&aes_instance);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the encryption process. */
    while (false == state) {
    }

    if ((ref_cipher_text_cbc[0] != output_data[0]) ||
        (ref_cipher_text_cbc[1] != output_data[1]) ||
        (ref_cipher_text_cbc[2] != output_data[2]) ||
        (ref_cipher_text_cbc[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }

    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CBC decipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;

```

```

g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_CBC_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector);

aes_set_new_message(&aes_instance);
/* Write the data to be deciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_cipher_text_cbc);
aes_clear_new_message(&aes_instance);

/* Wait for the end of the decryption process. */
while (false == state) {
}

if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||
    (ref_plain_text[2] != output_data[2]) ||
    (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
}

static void cfb128_mode_test(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CFB128 cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_MANUAL_START;
    g_aes_cfg.opmode = AES_CFB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance, AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector);

    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_set_new_message(&aes_instance);
    aes_start(&aes_instance);
    aes_clear_new_message(&aes_instance);

```

```

/* Wait for the end of the encryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_cipher_text_cfb128[0] != output_data[0]) ||
    (ref_cipher_text_cfb128[1] != output_data[1]) ||
    (ref_cipher_text_cfb128[2] != output_data[2]) ||
    (ref_cipher_text_cfb128[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

printf("\r\n-----\r\n");
printf("- 128bit cryptographic key\r\n");
printf("- CFB128 decipher mode\r\n");
printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_MANUAL_START;
g_aes_cfg.opmode = AES_CFB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector);

/* Write the data to be deciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_cipher_text_cfb128);
aes_set_new_message(&aes_instance);
aes_start(&aes_instance);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the decryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||
    (ref_plain_text[2] != output_data[2]) ||
    (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
}

static void ofb_mode_test(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- OFB cipher mode\r\n");

```

```

printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_OFB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector);
aes_set_new_message(&aes_instance);
/* Write the data to be ciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_plain_text);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the encryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_cipher_text_ofb[0] != output_data[0]) ||
    (ref_cipher_text_ofb[1] != output_data[1]) ||
    (ref_cipher_text_ofb[2] != output_data[2]) ||
    (ref_cipher_text_ofb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

printf("\r\n-----\r\n");
printf("- 128bit cryptographic key\r\n");
printf("- OFB decipher mode\r\n");
printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_MANUAL_START;
g_aes_cfg.opmode = AES_OFB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector);

/* Write the data to be deciphered to the input data registers. */

```



```

aes_write_input_data(&aes_instance, ref_cipher_text_ofb);

aes_set_new_message(&aes_instance);
aes_start(&aes_instance);
aes_clear_new_message(&aes_instance);

/* Wait for the end of the decryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||
    (ref_plain_text[2] != output_data[2]) ||
    (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
}

static void ctr_mode_test(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CTR cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_CTR_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance, AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector_ctr);
    aes_set_new_message(&aes_instance);
    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_clear_new_message(&aes_instance);

    /* Wait for the end of the encryption process. */
    while (false == state) {
    }

    /* check the result. */
    if ((ref_cipher_text_ctr[0] != output_data[0]) ||
        (ref_cipher_text_ctr[1] != output_data[1]) ||
        (ref_cipher_text_ctr[2] != output_data[2]) ||
        (ref_cipher_text_ctr[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
}

```

```

}

printf("\r\n-----\r\n");
printf("- 128bit cryptographic key\r\n");
printf("- CTR decipher mode\r\n");
printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-----\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_MANUAL_START;
g_aes_cfg.opmode = AES_CTR_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector_ctr);

/* Write the data to be deciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_cipher_text_ctr);

aes_set_new_message(&aes_instance);
aes_start(&aes_instance);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the decryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_plain_text[0] != output_data[0]) ||
    (ref_plain_text[1] != output_data[1]) ||
    (ref_plain_text[2] != output_data[2]) ||
    (ref_plain_text[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}
}

```

Add to user application initialization (typically the start of `main()`):

```

/* Initialize the system and console*/
system_init();
configure_usart();

aes_get_config_defaults(&g_aes_cfg);
aes_init(&aes_instance, AES, &g_aes_cfg);

aes_enable(&aes_instance);

/* Enable AES interrupt. */
aes_register_callback(aes_callback, AES_CALLBACK_ENCRYPTION_COMPLETE);
aes_enable_callback(&aes_instance, AES_CALLBACK_ENCRYPTION_COMPLETE);

```

Workflow

1. Define sample data from NIST-800-38A appendix F for ECB mode.

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

uint32_t ref_cipher_text_cbc[AES_EXAMPLE_REFBUF_SIZE] = {
    0xacab4976,
    0x46b21981,
    0x9b8ee9ce,
    0x7d19e912
};

uint32_t ref_cipher_text_cfb128[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ofb[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ctr[AES_EXAMPLE_REFBUF_SIZE] = {
    0x91614d87,
    0x26e320b6,
    0x6468ef1b,
    0xceb60d99
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};

const uint32_t init_vector[4] = {
    0x03020100,
    0x07060504,
    0x0b0a0908,
    0x0f0e0d0c
};

const uint32_t init_vector_ctr[4] = {
```

```

        0xf3f2f1f0,
        0xf7f6f5f4,
        0xfbfaf9f8,
        0xffffefdfc
    };

    /* @} */

```

2. Create related module variable and software instance structure.

```

/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;

struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;

```

3. Configure, initialize, and enable AES module.

1. Configuration AES struct, which can be filled out to adjust the configuration of a physical AES peripheral.

```

aes_get_config_defaults(&g_aes_cfg);

```

2. Initialize the AES configuration struct with the module's default values.

```

aes_init(&aes_instance, AES, &g_aes_cfg);

```

3. Enable the AES module.

```

aes_enable(&aes_instance);

```

4. Register and enable the AES module callback.

```

/* Enable AES interrupt. */
aes_register_callback(aes_callback, AES_CALLBACK_ENCRYPTION_COMPLETE);
aes_enable_callback(&aes_instance, AES_CALLBACK_ENCRYPTION_COMPLETE);

```

4.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```

ecb_mode_test();
cbc_mode_test();
cfb128_mode_test();
ofb_mode_test();
ctr_mode_test();

```

Workflow

1. Configure ECB mode encryption and decryption and run test.

```

ecb_mode_test();

```

2. Configure CBC mode encryption and decryption and run test.

```

cbc_mode_test();

```

3. Configure CFB mode encryption and decryption and run test.

```
cfb128_mode_test();
```

4. Configure OFB mode encryption and decryption and run test.

```
ofb_mode_test();
```

5. Configure CTR mode encryption and decryption and run test.

```
ctr_mode_test();
```

4.8.3. Quick Start Guide for AES - DMA

The supported board list:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro

This example demonstrates how to use the AES driver to perform:

- ECB encryption with DMA

Upon startup, the program uses the USART driver to display application output message from which ECB DMA encryption modes can be tested.

4.8.3.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, outside of any functions:

```
#define AES_EXAMPLE_REFBUF_SIZE 4

uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};
```

Add to the main application source file, outside of any functions:

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;
```

```

struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;

```

Copy-paste the following setup code to your user application:

```

static void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}

static void ecb_mode_test_dma(void)
{
    printf("\r\n-----\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB cipher mode\r\n");
    printf("- DMA mode\r\n");
    printf("- 4 32bit words with DMA\r\n");
    printf("-----\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance, AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    dma_start_transfer_job(&example_resource_tx);
    aes_set_new_message(&aes_instance);
    aes_clear_new_message(&aes_instance);

    /* Wait DMA transfer */
    while (false == state) {
    }

    /* Wait for the end of the encryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
    }

    state = false;
    dma_start_transfer_job(&example_resource_rx);

    /* Wait DMA transfer */
}

```

```

while (false == state) {
}

if ((ref_cipher_text_ecb[0] != output_data[0]) ||
    (ref_cipher_text_ecb[1] != output_data[1]) ||
    (ref_cipher_text_ecb[2] != output_data[2]) ||
    (ref_cipher_text_ecb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

}

static void transfer_tx_rx_done(struct dma_resource* const resource )
{
    state = true;
}

static void configure_dma_aes_wr(void)
{
    struct dma_resource_config tx_config;
    dma_get_config_defaults(&tx_config);

    tx_config.peripheral_trigger = AES_DMAC_ID_WR;
    tx_config.trigger_action = DMA_TRIGGER_ACTION_BLOCK;

    dma_allocate(&example_resource_tx, &tx_config);

    struct dma_descriptor_config tx_descriptor_config;

    dma_descriptor_get_config_defaults(&tx_descriptor_config);

    tx_descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
    tx_descriptor_config.dst_increment_enable = false;
    tx_descriptor_config.block_transfer_count = AES_EXAMPLE_REFBUF_SIZE;
    tx_descriptor_config.source_address = (uint32_t)ref_plain_text +
sizeof(ref_plain_text);
    tx_descriptor_config.destination_address = (uint32_t) &(AES->INDATA);
    dma_descriptor_create(&example_descriptor_tx, &tx_descriptor_config);

    dma_add_descriptor(&example_resource_tx, &example_descriptor_tx);
}

static void configure_dma_aes_rd(void)
{
    struct dma_resource_config rx_config;

    dma_get_config_defaults(&rx_config);

    rx_config.peripheral_trigger = AES_DMAC_ID_RD;
    rx_config.trigger_action = DMA_TRIGGER_ACTION_BLOCK;

    dma_allocate(&example_resource_rx, &rx_config);

    struct dma_descriptor_config rx_descriptor_config;

    dma_descriptor_get_config_defaults(&rx_descriptor_config);

    rx_descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
    rx_descriptor_config.src_increment_enable = false;
    rx_descriptor_config.block_transfer_count = AES_EXAMPLE_REFBUF_SIZE;

```

```

rx_descriptor_config.source_address = (uint32_t)&(AES->INDATA);
rx_descriptor_config.destination_address =
    (uint32_t)output_data + sizeof(output_data);

dma_descriptor_create(&example_descriptor_rx, &rx_descriptor_config);

dma_add_descriptor(&example_resource_rx, &example_descriptor_rx);
}

```

Add to user application initialization (typically the start of `main()`):

```

/* Initialize the system and console*/
system_init();
configure_usart();

/* Configure AES DMA and enable callback */
configure_dma_aes_wr();
configure_dma_aes_rd();

dma_register_callback(&example_resource_tx, transfer_tx_rx_done,
    DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_tx, DMA_CALLBACK_TRANSFER_DONE);

dma_register_callback(&example_resource_rx, transfer_tx_rx_done,
    DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_rx, DMA_CALLBACK_TRANSFER_DONE);

aes_get_config_defaults(&g_aes_cfg);
aes_init(&aes_instance, AES, &g_aes_cfg);

aes_enable(&aes_instance);

```

Workflow

1. Define sample data from NIST-800-38A appendix F for ECB mode.

```

#define AES_EXAMPLE_REFBUF_SIZE 4

uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};

```


2. Create related module variable and software instance structure.

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;
struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;
```

3. Create DMA resource struct and descriptor.

```
struct dma_resource example_resource_tx;
struct dma_resource example_resource_rx;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_tx SECTION_DMAC_DESCRIPTOR;
DmacDescriptor example_descriptor_rx SECTION_DMAC_DESCRIPTOR;
```

4. Configure, initialize, and enable AES module.

1. Configuration AES DMA module, which can be used for AES.

```
/* Configure AES DMA and enable callback */
configure_dma_aes_wr();
configure_dma_aes_rd();

dma_register_callback(&example_resource_tx, transfer_tx_rx_done,
DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_tx,
DMA_CALLBACK_TRANSFER_DONE);

dma_register_callback(&example_resource_rx, transfer_tx_rx_done,
DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_rx,
DMA_CALLBACK_TRANSFER_DONE);
```

2. Configuration AES struct, which can be filled out to adjust the configuration of a physical AES peripheral.

```
aes_get_config_defaults(&g_aes_cfg);
```

3. Initialize the AES configuration struct with the module's default values.

```
aes_init(&aes_instance, AES, &g_aes_cfg);
```

4. Enable the AES module.

```
aes_enable(&aes_instance);
```

4.8.3.2. Use Case

Code

Copy-paste the following code to your user application:

```
/* ECB mode encryption test with DMA */
ecb_mode_test_dma();
```

Workflow

1. Configure ECB mode encryption with DMA and run test.

```
state = false;
```

```

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_ECB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance, AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* The initialization vector is not used by the ECB cipher mode. */

dma_start_transfer_job(&example_resource_tx);
aes_set_new_message(&aes_instance);
aes_clear_new_message(&aes_instance);

/* Wait DMA transfer */
while (false == state) {
}

/* Wait for the end of the encryption process. */
while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
}

state = false;
dma_start_transfer_job(&example_resource_rx);

/* Wait DMA transfer */
while (false == state) {
}

if ((ref_cipher_text_ecb[0] != output_data[0]) ||
    (ref_cipher_text_ecb[1] != output_data[1]) ||
    (ref_cipher_text_ecb[2] != output_data[2]) ||
    (ref_cipher_text_ecb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

```

5. SAM Brown Out Detector (BOD) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Brown Out Detector (BOD) modules, to detect and respond to under-voltage events and take an appropriate action.

The following peripheral is used by this module:

- SUPC (Supply Controller)

The following devices can use this module:

- Atmel | SMART SAM L21/L22

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

5.1. Prerequisites

There are no prerequisites for this module.

5.2. Module Overview

The SAM devices contain a number of Brown Out Detector (BOD) modules. Each BOD monitors the supply voltage for any dips that go below the set threshold for the module. In case of a BOD detection the BOD will either reset the system or raise a hardware interrupt so that a safe power-down sequence can be attempted.

5.3. Special Considerations

The time between a BOD interrupt being raised and a failure of the processor to continue executing (in the case of a core power failure) is system specific; care must be taken that all critical BOD detection events can complete within the amount of time available.

5.4. Extra Information

For extra information, see [Extra Information for BOD Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

5.5. Examples

For a list of examples related to this driver, see [Examples for BOD Driver](#).

5.6. API Overview

5.6.1. Structure Definitions

5.6.1.1. Struct `bod12_config`

Configuration structure for a BOD12 module.

Table 5-1. Members

Type	Name	Description
enum bod12_action	action	Action to perform when a low power detection is made
bool	hysteresis	If <code>true</code> , enables detection hysteresis
uint8_t	level	BOD12 level to trigger at (see electrical section of device datasheet)
enum bod12_mode_in_active	mode_in_active	BOD12 configuration in active mode
enum bod12_mode_in_standby	mode_in_standby	BOD12 configuration in backup sleep mode
enum bod12_prescale	prescaler	Input sampler clock prescaler factor, to reduce the 1KHz clock from the ULP32K to lower the sampling rate of the BOD12
bool	run_in_standby	If <code>true</code> , the BOD12 is kept enabled and sampled during device sleep

5.6.1.2. Struct `bod33_config`

Configuration structure for a BOD33 module.

Table 5-2. Members

Type	Name	Description
enum bod33_action	action	Action to perform when a low power detection is made
uint8_t	backuplevel	BOD33 level to trigger at when monitors VBAT or in backup sleep mode
bool	hysteresis	If <code>true</code> , enables detection hysteresis
uint8_t	level	BOD33 level to trigger at when monitors VDD except in backup sleep mode
enum bod33_mode_in_active	mode_in_active	BOD33 configuration in active mode

Type	Name	Description
enum bod33_mode_in_standby	mode_in_standby	BOD33 configuration in backup sleep mode
enum bod33_vol_monitor	monitor	Voltage monitored in active and standby mode
enum bod33_prescale	prescaler	Input sampler clock prescaler factor, to reduce the 1KHz clock from the ULP32K to lower the sampling rate of the BOD33
bool	run_in_backup	If <code>true</code> , the BOD33 is kept enabled and sampled during device sleep
bool	run_in_standby	If <code>true</code> , the BOD33 is kept enabled and sampled during standby

5.6.2. Function Definitions

5.6.2.1. Configuration and Initialization

Function `bod33_get_config_defaults()`

Get default BOD33 configuration.

```
void bod33_get_config_defaults(
    struct bod33_config *const conf)
```

The default BOD33 configuration is:

- Clock prescaler set to divide the input clock by two
- Continuous in active mode
- Continuous in standby mode
- Monitor the V_{DD} power pin
- No action on BOD33 detect
- Hysteresis enabled
- BOD33 level 0x7 on V_{DD}
- BOD33 level 0x7 on V_{BAT}
- BOD33 kept enabled during device sleep
- BOD33 kept enabled during standby

Table 5-3. Parameters

Data direction	Parameter name	Description
[out]	conf	BOD33 configuration struct to set to default settings

Function `bod33_set_config()`

Configure a Brown Out Detector module.

```
enum status_code bod33_set_config(
    struct bod33_config *const conf)
```

Configures a given BOD module with the settings stored in the given configuration structure.

Table 5-4. Parameters

Data direction	Parameter name	Description
[in]	conf	Configuration settings to use for the specified BOD33

Table 5-5. Return Values

Return value	Description
STATUS_OK	Operation completed successfully
STATUS_ERR_INVALID_ARG	An invalid BOD was supplied
STATUS_ERR_INVALID_OPTION	The requested BOD level was outside the acceptable range

Function bod33_enable()

Enables a configured BOD33 module.

```
enum status_code bod33_enable( void )
```

Enables the BOD33 module that has been previously configured.

Returns

Error code indicating the status of the enable operation.

Table 5-6. Return Values

Return value	Description
STATUS_OK	If the BOD33 was successfully enabled

Function bod33_disable()

Disables an enabled BOD33 module.

```
enum status_code bod33_disable( void )
```

Disables the BOD33 module that was previously enabled.

Returns

Error code indicating the status of the disable operation.

Table 5-7. Return Values

Return value	Description
STATUS_OK	If the BOD33 was successfully disabled

Function bod33_is_detected()

Checks if the BOD33 low voltage detection has occurred.

```
bool bod33_is_detected( void )
```

Determines if the BOD33 has detected a voltage lower than its configured threshold.

Returns

Detection status of the BOD33.

Table 5-8. Return Values

Return value	Description
true	If the BOD33 has detected a low voltage condition
false	If the BOD33 has not detected a low voltage condition

Function bod33_clear_detected()

Clears the low voltage detection state of the BOD33.

```
void bod33_clear_detected( void )
```

Clears the low voltage condition of BOD33 module, so that new low voltage conditions can be detected.

Function bod12_get_config_defaults()

Get default BOD12 configuration.

```
void bod12_get_config_defaults(  
    struct bod12_config *const conf)
```

The default BOD12 configuration is:

- Clock prescaler set to divide the input clock by two
- Continuous in active mode
- Continuous in standby mode
- Reset on BOD12 detect
- Hysteresis enabled
- BOD12 level 0x12
- BOD12 kept enabled during device sleep

Table 5-9. Parameters

Data direction	Parameter name	Description
[out]	conf	BOD12 configuration struct to set to default settings

Function bod12_set_config()

Configure a Brown Out Detector module.

```
enum status_code bod12_set_config(  
    struct bod12_config *const conf)
```

Configures a given BOD module with the settings stored in the given configuration structure.

Table 5-10. Parameters

Data direction	Parameter name	Description
[in]	conf	Configuration settings to use for the specified BOD12

Table 5-11. Return Values

Return value	Description
STATUS_OK	Operation completed successfully
STATUS_ERR_INVALID_ARG	An invalid BOD was supplied
STATUS_ERR_INVALID_OPTION	The requested BOD level was outside the acceptable range

Function bod12_enable()

Enables a configured BOD12 module.

```
enum status_code bod12_enable( void )
```

Enables the BOD12 module that has been previously configured.

Returns

Error code indicating the status of the enable operation.

Table 5-12. Return Values

Return value	Description
STATUS_OK	If the BOD12 was successfully enabled

Function bod12_disable()

Disables an enabled BOD12 module.

```
enum status_code bod12_disable( void )
```

Disables the BOD12 module that was previously enabled.

Returns

Error code indicating the status of the disable operation.

Table 5-13. Return Values

Return value	Description
STATUS_OK	If the BOD12 was successfully disabled

Function bod12_is_detected()

Checks if the BOD12 low voltage detection has occurred.

```
bool bod12_is_detected( void )
```

Determines if the BOD12 has detected a voltage lower than its configured threshold.

Returns

Detection status of the BOD12.

Table 5-14. Return Values

Return value	Description
true	If the BOD12 has detected a low voltage condition
false	If the BOD12 has not detected a low voltage condition

Function bod12_clear_detected()

Clears the low voltage detection state of the BOD12.

```
void bod12_clear_detected( void )
```

Clears the low voltage condition of BOD12 module, so that new low voltage conditions can be detected.

5.6.3. Enumeration Definitions**5.6.3.1. Enum bod12_action**

List of possible BOD12 actions when a BOD12 module detects a brown-out condition.

Table 5-15. Members

Enum value	Description
BOD12_ACTION_NONE	A BOD12 detect will do nothing, and the BOD12 state must be polled
BOD12_ACTION_RESET	A BOD12 detect will reset the device
BOD12_ACTION_INTERRUPT	A BOD12 detect will fire an interrupt

5.6.3.2. Enum bod12_mode_in_active

List of possible BOD12 module voltage sampling modes in active sleep mode.

Table 5-16. Members

Enum value	Description
BOD12_ACTCFG_CONTINUOUS	BOD12 will sample the supply line continuously
BOD12_ACTCFG_SAMPLED	BOD12 will use the BOD12 sampling clock (1KHz) to sample the supply line

5.6.3.3. Enum bod12_mode_in_standby

List of possible BOD12 module voltage sampling modes in standby sleep mode.

Table 5-17. Members

Enum value	Description
BOD12_STDBYCFG_CONTINUOUS	BOD12 will sample the supply line continuously
BOD12_STDBYCFG_SAMPLED	BOD12 will use the BOD12 sampling clock (1KHz) to sample the supply line

5.6.3.4. Enum bod12_prescale

List of possible BOD12 controller prescaler values, to reduce the sampling speed of a BOD12 to lower the power consumption.

Table 5-18. Members

Enum value	Description
BOD12_PRESCALE_DIV_2	Divide input prescaler clock by 2
BOD12_PRESCALE_DIV_4	Divide input prescaler clock by 4
BOD12_PRESCALE_DIV_8	Divide input prescaler clock by 8
BOD12_PRESCALE_DIV_16	Divide input prescaler clock by 16
BOD12_PRESCALE_DIV_32	Divide input prescaler clock by 32
BOD12_PRESCALE_DIV_64	Divide input prescaler clock by 64
BOD12_PRESCALE_DIV_128	Divide input prescaler clock by 128
BOD12_PRESCALE_DIV_256	Divide input prescaler clock by 256
BOD12_PRESCALE_DIV_512	Divide input prescaler clock by 512
BOD12_PRESCALE_DIV_1024	Divide input prescaler clock by 1024
BOD12_PRESCALE_DIV_2048	Divide input prescaler clock by 2048
BOD12_PRESCALE_DIV_4096	Divide input prescaler clock by 4096
BOD12_PRESCALE_DIV_8192	Divide input prescaler clock by 8192
BOD12_PRESCALE_DIV_16384	Divide input prescaler clock by 16384
BOD12_PRESCALE_DIV_32768	Divide input prescaler clock by 32768
BOD12_PRESCALE_DIV_65536	Divide input prescaler clock by 65536

5.6.3.5. Enum bod33_action

List of possible BOD33 actions when a BOD33 module detects a brown-out condition.

Table 5-19. Members

Enum value	Description
BOD33_ACTION_NONE	A BOD33 detect will do nothing, and the BOD33 state must be polled
BOD33_ACTION_RESET	A BOD33 detect will reset the device
BOD33_ACTION_INTERRUPT	A BOD33 detect will fire an interrupt
BOD33_ACTION_BACKUP	A BOD33 detect will put the device in backup sleep mode

5.6.3.6. Enum bod33_mode_in_active

List of possible BOD33 module voltage sampling modes in active sleep mode.

Table 5-20. Members

Enum value	Description
BOD33_ACTCFG_CONTINUOUS	BOD33 will sample the supply line continuously
BOD33_ACTCFG_SAMPLED	BOD33 will use the BOD33 sampling clock (1KHz) to sample the supply line

5.6.3.7. Enum bod33_mode_in_standby

List of possible BOD33 module voltage sampling modes in standby sleep mode.

Table 5-21. Members

Enum value	Description
BOD33_STDBYCFG_CONTINUOUS	BOD33 will sample the supply line continuously
BOD33_STDBYCFG_SAMPLED	BOD33 will use the BOD33 sampling clock (1KHz) to sample the supply line

5.6.3.8. Enum bod33_prescale

List of possible BOD33 controller prescaler values, to reduce the sampling speed of a BOD33 to lower the power consumption.

Table 5-22. Members

Enum value	Description
BOD33_PRESCALE_DIV_2	Divide input prescaler clock by 2
BOD33_PRESCALE_DIV_4	Divide input prescaler clock by 4
BOD33_PRESCALE_DIV_8	Divide input prescaler clock by 8
BOD33_PRESCALE_DIV_16	Divide input prescaler clock by 16
BOD33_PRESCALE_DIV_32	Divide input prescaler clock by 32
BOD33_PRESCALE_DIV_64	Divide input prescaler clock by 64
BOD33_PRESCALE_DIV_128	Divide input prescaler clock by 128
BOD33_PRESCALE_DIV_256	Divide input prescaler clock by 256
BOD33_PRESCALE_DIV_512	Divide input prescaler clock by 512
BOD33_PRESCALE_DIV_1024	Divide input prescaler clock by 1024
BOD33_PRESCALE_DIV_2048	Divide input prescaler clock by 2048
BOD33_PRESCALE_DIV_4096	Divide input prescaler clock by 4096
BOD33_PRESCALE_DIV_8192	Divide input prescaler clock by 8192
BOD33_PRESCALE_DIV_16384	Divide input prescaler clock by 16384

Enum value	Description
BOD33_PRESCALE_DIV_32768	Divide input prescaler clock by 32768
BOD33_PRESCALE_DIV_65536	Divide input prescaler clock by 65536

5.6.3.9. Enum `bod33_vol_monitor`

List of possible BOD33 module voltage monitored in active and standby mode.

Table 5-23. Members

Enum value	Description
BOD33_VMON_VDD	The BOD33 monitors the VDD power pin in active and standby mode
BOD33_VMON_VBAT	The BOD33 monitors the VBAT power pin in active and standby mode

5.7. Extra Information for BOD Driver

5.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Definition
BOD	Brown Out Detector

5.7.2. Dependencies

This driver has the following dependencies:

- None

5.7.3. Errata

There are no errata related to this driver.

5.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

5.8. Examples for BOD Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Brown Out Detector \(BOD\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for BOD - Basic](#)
- [Application Use Case for BOD - Application](#)

5.8.1. Quick Start Guide for BOD - Basic

In this use case, the BOD33 and BOD12 will be configured with the following settings:

- Continuous sampling mode
- Prescaler setting of 2
- Reset action on low voltage detect

5.8.1.1. Quick Start

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
static void configure_bod33(void)
{
    struct bod33_config config_bod33;
    bod33_get_config_defaults(&config_bod33);

    bod33_set_config(&config_bod33);

    bod33_enable();
}

static void configure_bod12(void)
{
    struct bod12_config config_bod12;
    bod12_get_config_defaults(&config_bod12);

    bod12_set_config(&config_bod12);

    bod12_enable();
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_bod33();
```

```
configure_bod12();
```

Workflow

1. Create a BOD33 module configuration struct, which can be filled out to adjust the configuration of a physical BOD peripheral.

```
struct bod33_config config_bod33;
```

2. Initialize the BOD33 configuration struct with the module's default values.

```
bod33_get_config_defaults(&config_bod33);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Configure the BOD33 module with the desired settings.

```
bod33_set_config(&config_bod33);
```

4. Enable the BOD33 module so that it will monitor the power supply voltage.

```
bod33_enable();
```

The workflow of the BOD12 is the same as for the BOD33.

5.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    /* Infinite loop */  
}
```

Workflow

1. Enter an infinite loop so that the BOD can continue to monitor the supply voltage level.

```
while (true) {  
    /* Infinite loop */  
}
```

5.8.2. Application Use Case for BOD - Application

The preferred method of setting BOD33 levels and settings is through the fuses. When it is desirable to set it in software, see the below use case.

In this use case, a new BOD33 level might be set in SW if the clock settings are adjusted up after a battery has charged to a higher level. When the battery discharges, the chip will reset when the battery level is below SW BOD33 level. Now the chip will run at a lower clock rate and the BOD33 level from fuse. The chip should always measure the voltage before adjusting the frequency up.

6. SAM Configurable Custom Logic (CCL) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Configurable Custom Logic functionality.

The following peripheral is used by this module:

- CCL (Configurable Custom Logic)

The following devices can use this module:

- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

6.1. Prerequisites

There are no prerequisites for this module.

6.2. Module Overview

This driver provides an interface for the Configurable Custom Logic functions on the device.

The Configurable Custom Logic (CCL) contains programmable logic which can be connected to the device pins, events, or internal peripherals.

Each LUT consists of three inputs, a truth table and optional synchronizer, filter and edge detector. Each LUT can generate an output as a user programmable logic expression with three inputs.

The output can be combinatorially generated from the inputs, or filtered to remove spike. An optional sequential module can be enabled. The inputs of sequential module are individually controlled by two independent, adjacent LUT(LUT0/LUT1, LUT2/LUT3 etc.) outputs, enabling complex waveform generation.

6.3. Special Considerations

There are no special considerations for this module.

6.4. Extra Information

For extra information, see [Extra Information for CCL Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)

- [Errata](#)
- [Module History](#)

6.5. Examples

For a list of examples related to this driver, see [Examples for CCL Driver](#).

6.6. API Overview

6.6.1. Structure Definitions

6.6.1.1. Struct `ccl_config`

Configuration structure for CCL module.

Table 6-1. Members

Type	Name	Description
enum <code>gclk_generator</code>	<code>clock_source</code>	GCLK generator used to clock the peripheral
bool	<code>run_in_standby</code>	If <code>true</code> , the GCLK_CCL clock will not stop in standby sleep mode

6.6.1.2. Struct `ccl_lut_config`

Configuration structure for CCL LUT 0 to 3.

Table 6-2. Members

Type	Name	Description
bool	<code>edge_selection_enable</code>	If <code>true</code> , Edge detector is enabled
bool	<code>event_input_enable</code>	If <code>true</code> , LUT incoming event is enabled
bool	<code>event_input_inverted_enable</code>	If <code>true</code> , incoming event is inverted
bool	<code>event_output_enable</code>	If <code>true</code> , LUT event output is enabled
enum <code>ccl_lut_filter_sel</code>	<code>filter_sel</code>	Selection of the LUT output filter options
enum <code>ccl_lut_input_src_sel</code>	<code>input0_src_sel</code>	Selection of the input0 source
enum <code>ccl_lut_input_src_sel</code>	<code>input1_src_sel</code>	Selection of the input1 source
enum <code>ccl_lut_input_src_sel</code>	<code>input2_src_sel</code>	Selection of the input2 source
uint8_t	<code>truth_table_value</code>	The value of truth logic as a function of inputs IN[2:0]

6.6.2. Function Definitions

6.6.2.1. Initialize and Reset CCL Module

Function `ccl_init()`

Initializes CCL module.

```
void ccl_init(  
    struct ccl_config *const config)
```

Resets all registers in the MODULE to their initial state, and then enable the module.

Function `ccl_get_config_defaults()`

Initializes all members of a CCL configuration structure to safe defaults.

```
void ccl_get_config_defaults(  
    struct ccl_config *const config)
```

Initializes all members of a given Configurable Custom Logic configuration structure to safe and known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- GCLK_CLL will be stopped in standby sleep mode
- Generator 0 is the default GCLK generator

Table 6-3. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `ccl_module_reset()`

Resets CCL module.

```
void ccl_module_reset( void )
```

Resets all registers in the MODULE to their initial state, and the CCL will be disabled.

6.6.2.2. Enable and Disable CCL Module

Function `ccl_module_enable()`

Enables CCL module.

```
void ccl_module_enable( void )
```

Enable the peripheral.

Function `ccl_module_disable()`

Disables CCL module.

```
void ccl_module_disable( void )
```

Disables the peripheral.

6.6.2.3. Configure LUT

Function `ccl_seq_config()`

Writes sequential selection to the hardware module.

```
enum status_code ccl_seq_config(  
    const enum ccl_seq_id number,  
    const enum ccl_seq_selection seq_selection)
```

Writes a given sequential selection configuration to the hardware module.

Note: This function can only be used when the CCL module is disabled.

Table 6-4. Parameters

Data direction	Parameter name	Description
[in]	seq_selection	Enum for the sequential selection configuration
[in]	number	SEQ unit number to config

Function `ccl_lut_get_config_defaults()`

Initializes all members of LUT configuration structure to safe defaults.

```
void ccl_lut_get_config_defaults(  
    struct ccl_lut_config *const config)
```

Initializes all members of LUT configuration structure to safe defaults. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Truth table value is 0x00
- LUT event output is disabled
- LUT incoming event is disabled
- LUT incoming event is not inverted
- The input IN[2:0] source is masked
- The edge detector is disabled
- The LUT output filter is disabled

Table 6-5. Parameters

Data direction	Parameter name	Description
[out]	config	LUT configuration structure to initialize to default values

Function `ccl_lut_set_config()`

Writes LUT configuration to the hardware module.

```
enum status_code ccl_lut_set_config(  
    const enum ccl_lut_id number,  
    struct ccl_lut_config *const config)
```

Writes a given LUT configuration to the hardware module.

Note: This function can only be used when the CCL module is disabled.

Table 6-6. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to the LUT configuration struct
[in]	number	LUT number to config

6.6.2.4. Enable and Disable LUT**Function ccl_lut_enable()**

Enables an LUT that was previously configured.

```
void ccl_lut_enable(
    const enum ccl_lut_id number)
```

Enables an LUT that was previously configured via a call to [ccl_lut_set_config](#) function.

Table 6-7. Parameters

Data direction	Parameter name	Description
[in]	number	LUT number to enable

Function ccl_lut_disable()

Disables an LUT that was previously enabled.

```
void ccl_lut_disable(
    const enum ccl_lut_id number)
```

Disables an LUT that was previously enabled via a call to [ccl_lut_enable\(\)](#).

Table 6-8. Parameters

Data direction	Parameter name	Description
[in]	number	LUT number to enable

6.6.3. Enumeration Definitions**6.6.3.1. Enum ccl_lut_filter_sel**

Enum for the LUT output filter options.

Table 6-9. Members

Enum value	Description
CCL_LUT_FILTER_DISABLE	Filter disabled
CCL_LUT_FILTER_SYNC	Synchronizer enabled
CCL_LUT_FILTER_ENABLE	Filter enabled

6.6.3.2. Enum ccl_lut_id

Table 6-10. Members

Enum value	Description
CCL_LUT_0	CCL LUT 0
CCL_LUT_1	CCL LUT 1
CCL_LUT_2	CCL LUT 2
CCL_LUT_3	CCL LUT 3

6.6.3.3. Enum ccl_lut_input_src_sel

Enum for the LUT Input source selection.

Table 6-11. Members

Enum value	Description
CCL_LUT_INPUT_SRC_MASK	Masked input
CCL_LUT_INPUT_SRC_FEEDBACK	Feedback input source
CCL_LUT_INPUT_SRC_LINK	Linked LUT input source
CCL_LUT_INPUT_SRC_EVENT	Event input source
CCL_LUT_INPUT_SRC_IO	I/O pin input source
CCL_LUT_INPUT_SRC_AC	AC input source
CCL_LUT_INPUT_SRC_TC	TC input source
CCL_LUT_INPUT_SRC_ALTTTC	Alternative TC input source
CCL_LUT_INPUT_SRC_TCC	TCC input source
CCL_LUT_INPUT_SRC_SERCOM	SERCOM input source

6.6.3.4. Enum ccl_seq_id

Table 6-12. Members

Enum value	Description
CCL_SEQ_0	CCL SEQ 0
CCL_SEQ_1	CCL SEQ 1

6.6.3.5. Enum ccl_seq_selection

Enum for the sequential selection configuration.

Table 6-13. Members

Enum value	Description
CCL_SEQ_DISABLED	Sequential logic is disabled
CCL_SEQ_D_FLIP_FLOP	D flip flop
CCL_SEQ_JK_FLIP_FLOP	JK flip flop
CCL_SEQ_D_LATCH	D latch
CCL_SEQ_RS_LATCH	RS latch

6.7. Extra Information for CCL Driver

6.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
CCL	Configurable Custom Logic

6.7.2. Dependencies

This driver has no dependencies.

6.7.3. Errata

There are no errata related to this driver.

6.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

6.8. Examples for CCL Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Configurable Custom Logic \(CCL\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for CCL - Basic](#)

6.8.1. Quick Start Guide for CCL - Basic

In this use case, the LUT0 and LUT1 input source is configured as I/O pin. The LUT0 and LUT1 pair is connected to internal sequential logic, which is configured as D flip flop mode.

6.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_ccl(void)
{
    struct ccl_config conf;

    ccl_get_config_defaults(&conf);

    ccl_init(&conf);
}

void configure_ccl_lut0(void)
{
    struct ccl_lut_config conf;

    ccl_lut_get_config_defaults(&conf);

    conf.truth_table_value = 0x02;
    conf.input0_src_sel = CCL_LUT_INPUT_SRC_IO;
    conf.input1_src_sel = CCL_LUT_INPUT_SRC_IO;
    conf.input2_src_sel = CCL_LUT_INPUT_SRC_IO;
    conf.filter_sel = CCL_LUTCTRL_FILTSEL_FILTER;

    struct system_pinmux_config lut0_input_pin0_conf, lut0_input_pin1_conf,
    lut0_input_pin2_conf;
    system_pinmux_get_config_defaults(&lut0_input_pin0_conf);
    system_pinmux_get_config_defaults(&lut0_input_pin1_conf);
    system_pinmux_get_config_defaults(&lut0_input_pin2_conf);
    lut0_input_pin0_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
    lut0_input_pin0_conf.mux_position = CCL_LUT0_IN0_MUX;
    lut0_input_pin1_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
    lut0_input_pin1_conf.mux_position = CCL_LUT0_IN1_MUX;
    lut0_input_pin2_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
    lut0_input_pin2_conf.mux_position = CCL_LUT0_IN2_MUX;
    system_pinmux_pin_set_config(CCL_LUT0_IN0_PIN, &lut0_input_pin0_conf);
    system_pinmux_pin_set_config(CCL_LUT0_IN1_PIN, &lut0_input_pin1_conf);
    system_pinmux_pin_set_config(CCL_LUT0_IN2_PIN, &lut0_input_pin2_conf);
    struct system_pinmux_config lut0_out_pin_conf;
    system_pinmux_get_config_defaults(&lut0_out_pin_conf);
    lut0_out_pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_OUTPUT;
    lut0_out_pin_conf.mux_position = CCL_LUT0_OUT_MUX;
    system_pinmux_pin_set_config(CCL_LUT0_OUT_PIN, &lut0_out_pin_conf);

    ccl_lut_set_config(CCL_LUT_0, &conf);
}

void configure_ccl_lut1(void)
{
    struct ccl_lut_config conf;

    ccl_lut_get_config_defaults(&conf);

    conf.truth_table_value = 0x02;
```

```

    conf.input0_src_sel = CCL_LUT_INPUT_SRC_IO;
    conf.input1_src_sel = CCL_LUT_INPUT_SRC_IO;
    conf.input2_src_sel = CCL_LUT_INPUT_SRC_IO;
    conf.filter_sel = CCL_LUTCTRL_FILTSEL_FILTER;

    struct system_pinmux_config lut1_input_pin0_conf, lut1_input_pin1_conf,
    lut1_input_pin2_conf;
    system_pinmux_get_config_defaults(&lut1_input_pin0_conf);
    system_pinmux_get_config_defaults(&lut1_input_pin1_conf);
    system_pinmux_get_config_defaults(&lut1_input_pin2_conf);
    lut1_input_pin0_conf.direction      = SYSTEM_PINMUX_PIN_DIR_INPUT;
    lut1_input_pin0_conf.mux_position  = CCL_LUT1_IN0_MUX;
    lut1_input_pin1_conf.direction      = SYSTEM_PINMUX_PIN_DIR_INPUT;
    lut1_input_pin1_conf.mux_position  = CCL_LUT1_IN1_MUX;
    lut1_input_pin2_conf.direction      = SYSTEM_PINMUX_PIN_DIR_INPUT;
    lut1_input_pin2_conf.mux_position  = CCL_LUT1_IN2_MUX;
    system_pinmux_pin_set_config(CCL_LUT1_IN0_PIN, &lut1_input_pin0_conf);
    system_pinmux_pin_set_config(CCL_LUT1_IN1_PIN, &lut1_input_pin1_conf);
    system_pinmux_pin_set_config(CCL_LUT1_IN2_MUX, &lut1_input_pin2_conf);
    struct system_pinmux_config lut1_out_pin_conf;
    system_pinmux_get_config_defaults(&lut1_out_pin_conf);
    lut1_out_pin_conf.direction      = SYSTEM_PINMUX_PIN_DIR_OUTPUT;
    lut1_out_pin_conf.mux_position  = CCL_LUT1_OUT_MUX;
    system_pinmux_pin_set_config(CCL_LUT1_OUT_PIN, &lut1_out_pin_conf);

    ccl_lut_set_config(CCL_LUT_1, &conf);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_ccl();
configure_ccl_lut0();
configure_ccl_lut1();
ccl_seq_config(CCL_SEQ_0, CCL_SEQ_D_FLIP_FLOP);

ccl_lut_enable(CCL_LUT_0);
ccl_lut_enable(CCL_LUT_1);
ccl_module_enable();

```

Workflow

1. Creates a CCL configuration struct, which can be filled out to adjust the configuration of CCL.

```
struct ccl_config conf;
```

2. Settings and fill the CCL configuration struct with the default settings.

```
ccl_get_config_defaults(&conf);
```

3. Initializes CCL module.

```
ccl_init(&conf);
```

4. Creates a LUT configuration struct, which can be filled out to adjust the configuration of LUT0.

```
struct ccl_lut_config conf;
```

1. Fill the LUT0 configuration struct with the default settings.

```
ccl_lut_get_config_defaults(&conf);
```

5. Adjust the LUT0 configuration struct.

```

conf.truth_table_value = 0x02;
conf.input0_src_sel = CCL_LUT_INPUT_SRC_IO;

```

```

conf.input1_src_sel = CCL_LUT_INPUT_SRC_IO;
conf.input2_src_sel = CCL_LUT_INPUT_SRC_IO;
conf.filter_sel = CCL_LUTCTRL_FILTSEL_FILTER;

```

6. Set up LUT0 input and output pin.

```

struct system_pinmux_config lut0_input_pin0_conf, lut0_input_pin1_conf,
lut0_input_pin2_conf;
system_pinmux_get_config_defaults(&lut0_input_pin0_conf);
system_pinmux_get_config_defaults(&lut0_input_pin1_conf);
system_pinmux_get_config_defaults(&lut0_input_pin2_conf);
lut0_input_pin0_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
lut0_input_pin0_conf.mux_position = CCL_LUT0_IN0_MUX;
lut0_input_pin1_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
lut0_input_pin1_conf.mux_position = CCL_LUT0_IN1_MUX;
lut0_input_pin2_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
lut0_input_pin2_conf.mux_position = CCL_LUT0_IN2_MUX;
system_pinmux_pin_set_config(CCL_LUT0_IN0_PIN, &lut0_input_pin0_conf);
system_pinmux_pin_set_config(CCL_LUT0_IN1_PIN, &lut0_input_pin1_conf);
system_pinmux_pin_set_config(CCL_LUT0_IN2_PIN, &lut0_input_pin2_conf);
struct system_pinmux_config lut0_out_pin_conf;
system_pinmux_get_config_defaults(&lut0_out_pin_conf);
lut0_out_pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_OUTPUT;
lut0_out_pin_conf.mux_position = CCL_LUT0_OUT_MUX;
system_pinmux_pin_set_config(CCL_LUT0_OUT_PIN, &lut0_out_pin_conf);

```

7. Writes LUT0 configuration to the hardware module.

```

ccl_lut_set_config(CCL_LUT_0, &conf);

```

8. Creates a LUT configuration struct, which can be filled out to adjust the configuration of LUT1.

```

struct ccl_lut_config conf;

```

1. Fill the LUT1 configuration struct with the default settings.

```

ccl_lut_get_config_defaults(&conf);

```

9. Adjust the LUT1 configuration struct.

```

conf.truth_table_value = 0x02;
conf.input0_src_sel = CCL_LUT_INPUT_SRC_IO;
conf.input1_src_sel = CCL_LUT_INPUT_SRC_IO;
conf.input2_src_sel = CCL_LUT_INPUT_SRC_IO;
conf.filter_sel = CCL_LUTCTRL_FILTSEL_FILTER;

```

10. Set up LUT1 input and output pin.

```

struct system_pinmux_config lut1_input_pin0_conf, lut1_input_pin1_conf,
lut1_input_pin2_conf;
system_pinmux_get_config_defaults(&lut1_input_pin0_conf);
system_pinmux_get_config_defaults(&lut1_input_pin1_conf);
system_pinmux_get_config_defaults(&lut1_input_pin2_conf);
lut1_input_pin0_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
lut1_input_pin0_conf.mux_position = CCL_LUT1_IN0_MUX;
lut1_input_pin1_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
lut1_input_pin1_conf.mux_position = CCL_LUT1_IN1_MUX;
lut1_input_pin2_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
lut1_input_pin2_conf.mux_position = CCL_LUT1_IN2_MUX;
system_pinmux_pin_set_config(CCL_LUT1_IN0_PIN, &lut1_input_pin0_conf);
system_pinmux_pin_set_config(CCL_LUT1_IN1_PIN, &lut1_input_pin1_conf);
system_pinmux_pin_set_config(CCL_LUT1_IN2_MUX, &lut1_input_pin2_conf);
struct system_pinmux_config lut1_out_pin_conf;
system_pinmux_get_config_defaults(&lut1_out_pin_conf);
lut1_out_pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_OUTPUT;

```



```
lut1_out_pin_conf.mux_position = CCL_LUT1_OUT_MUX;  
system_pinmux_pin_set_config(CCL_LUT1_OUT_PIN, &lut1_out_pin_conf);
```

11. Writes LUT1 configuration to the hardware module.

```
ccl_lut_set_config(CCL_LUT_1, &conf);
```

12. Configure the sequential logic with the D flip flop mode.

```
ccl_seq_config(CCL_SEQ_0, CCL_SEQ_D_FLIP_FLOP);
```

6.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    /* Do nothing */  
}
```

7. SAM Direct Memory Access Controller (DMAC) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the Direct Memory Access Controller(DMAC) module within the device. The DMAC can transfer data between memories and peripherals, and thus off-load these tasks from the CPU. The module supports peripheral to peripheral, peripheral to memory, memory to peripheral, and memory to memory transfers.

The following peripheral is used by the DMAC Driver:

- DMAC (Direct Memory Access Controller)

The following devices can use this module:

- Atmel | SMART SAM D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

7.1. Prerequisites

There are no prerequisites for this module.

7.2. Module Overview

SAM devices with DMAC enables high data transfer rates with minimum CPU intervention and frees up CPU time. With access to all peripherals, the DMAC can handle automatic transfer of data to/from modules. It supports static and incremental addressing for both source and destination.

The DMAC when used with Event System or peripheral triggers, provides a considerable advantage by reducing the power consumption and performing data transfer in the background. For example, if the ADC is configured to generate an event, it can trigger the DMAC to transfer the data into another peripheral or SRAM. The CPU can remain in sleep during this time to reduce the power consumption.

Device	Dma channel number
SAM D21/R21/C20/C21	12
SAM D09/D10/D11	6
SAM L21,SAMR30	16

The DMA channel operation can be suspended at any time by software, by events from event system, or after selectable descriptor execution. The operation can be resumed by software or by events from the event system. The DMAC driver for SAM supports four types of transfers such as peripheral to peripheral, peripheral to memory, memory to peripheral, and memory to memory.

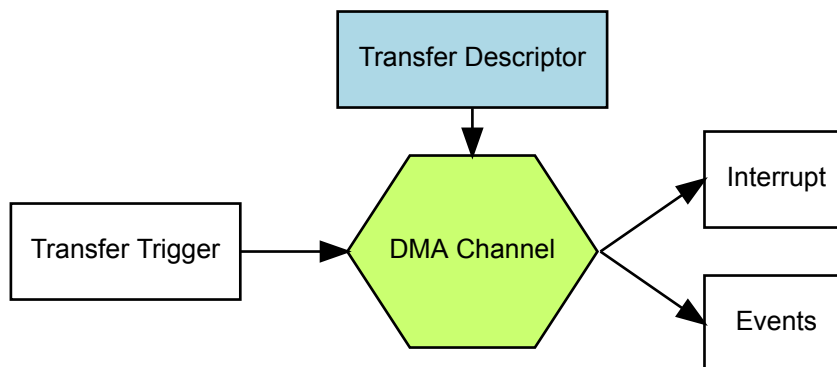
The basic transfer unit is a beat, which is defined as a single bus access. There can be multiple beats in a single block transfer and multiple block transfers in a DMA transaction. DMA transfer is based on descriptors, which holds transfer properties such as the source and destination addresses, transfer counter, and other additional transfer control information. The descriptors can be static or linked. When static, a single block transfer is performed. When linked, a number of transfer descriptors can be used to enable multiple block transfers within a single DMA transaction.

The implementation of the DMA driver is based on the idea that the DMA channel is a finite resource of entities with the same abilities. A DMA channel resource is able to move a defined set of data from a source address to destination address triggered by a transfer trigger. On the SAM devices there are 12 DMA resources available for allocation. Each of these DMA resources can trigger interrupt callback routines and peripheral events. The other main features are:

- Selectable transfer trigger source
 - Software
 - Event System
 - Peripheral
- Event input and output is supported for the four lower channels
- Four level channel priority
- Optional interrupt generation on transfer complete, channel error, or channel suspend
- Supports multi-buffer or circular buffer mode by linking multiple descriptors
- Beat size configurable as 8-bit, 16-bit, or 32-bit

A simplified block diagram of the DMA Resource can be seen in [Figure 7-1](#).

Figure 7-1. Module Overview



7.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_DMA_CHANNEL_STANDBY	SAM L21/L22/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

7.2.2. Terminology Used in DMAC Transfers

Name	Description
Beat	It is a single bus access by the DMAC. Configurable as 8-bit, 16-bit, or 32-bit.
Burst	It is a transfer of n-beats (n=1,4,8,16). For the DMAC module in SAM, the burst size is one beat. Arbitration takes place each time a burst transfer is completed.
Block transfer	A single block transfer is a configurable number of (1 to 64k) beat transfers

7.2.3. DMA Channels

The DMAC in each device consists of several DMA channels, which along with the transfer descriptors defines the data transfer properties.

- The transfer control descriptor defines the source and destination addresses, source and destination address increment settings, the block transfer count, and event output condition selection
- Dedicated channel registers control the peripheral trigger source, trigger mode settings, event input actions, and channel priority level settings

With a successful DMA resource allocation, a dedicated DMA channel will be assigned. The channel will be occupied until the DMA resource is freed. A DMA resource handle is used to identify the specific DMA resource. When there are multiple channels with active requests, the arbiter prioritizes the channels requesting access to the bus.

7.2.4. DMA Triggers

DMA transfer can be started only when a DMA transfer request is acknowledged/granted by the arbiter. A transfer request can be triggered from software, peripheral, or an event. There are dedicated source trigger selections for each DMA channel usage.

7.2.5. DMA Transfer Descriptor

The transfer descriptor resides in the SRAM and defines these channel properties.

Field name	Field width
Descriptor Next Address	32 bits
Destination Address	32 bits
Source Address	32 bits
Block Transfer Counter	16 bits
Block Transfer Control	16 bits

Before starting a transfer, at least one descriptor should be configured. After a successful allocation of a DMA channel, the transfer descriptor can be added with a call to [dma_add_descriptor\(\)](#). If there is a transfer descriptor already allocated to the DMA resource, the descriptor will be linked to the next descriptor address.

7.2.6. DMA Interrupts/Events

Both an interrupt callback and an peripheral event can be triggered by the DMA transfer. Three types of callbacks are supported by the DMA driver: transfer complete, channel suspend, and transfer error. Each of these callback types can be registered and enabled for each channel independently through the DMA driver API.

The DMAC module can also generate events on transfer complete. Event generation is enabled through the DMA channel, event channel configuration, and event user multiplexing is done through the events driver.

The DMAC can generate events in the below cases:

- When a block transfer is complete
- When each beat transfer within a block transfer is complete

7.3. Special Considerations

There are no special considerations for this module.

7.4. Extra Information

For extra information, see [Extra Information for DMAC Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

7.5. Examples

For a list of examples related to this driver, see [Examples for DMAC Driver](#).

7.6. API Overview

7.6.1. Variable and Type Definitions

7.6.1.1. Type dma_callback_t

```
typedef void(* dma_callback_t )(struct dma_resource *const resource)
```

Type definition for a DMA resource callback function.

7.6.1.2. Variable descriptor_section

```
DmacDescriptor descriptor_section
```

ExInitial description section.

7.6.1.3. Variable g_chan_interrupt_flag

```
uint8_t g_chan_interrupt_flag
```

7.6.2. Structure Definitions

7.6.2.1. Struct dma_descriptor_config

DMA transfer descriptor configuration. When the source or destination address increment is enabled, the addresses stored into the configuration structure must correspond to the end of the transfer.

Table 7-1. Members

Type	Name	Description
enum dma_beat_size	beat_size	Beat size is configurable as 8-bit, 16-bit, or 32-bit
enum dma_block_action	block_action	Action taken when a block transfer is completed
uint16_t	block_transfer_count	It is the number of beats in a block. This count value is decremented by one after each beat data transfer.
bool	descriptor_valid	Descriptor valid flag used to identify whether a descriptor is valid or not
uint32_t	destination_address	Transfer destination address
bool	dst_increment_enable	Used for enabling the destination address increment
enum dma_event_output_selection	event_output_selection	This is used to generate an event on specific transfer action in a channel. Supported only in four lower channels.
uint32_t	next_descriptor_address	Set to zero for static descriptors. This must have a valid memory address for linked descriptors.
uint32_t	source_address	Transfer source address
bool	src_increment_enable	Used for enabling the source address increment

Type	Name	Description
enum dma_step_selection	step_selection	This bit selects whether the source or destination address is using the step size settings
enum dma_address_increment_stepsize	step_size	The step size for source/destination address increment. The next address is calculated as $\text{next\_addr} = \text{addr} + (2^{\text{step\_size}} * \text{beat size})$.

7.6.2.2. Struct dma_events_config

Configurations for DMA events.

Table 7-2. Members

Type	Name	Description
bool	event_output_enable	Enable DMA event output
enum dma_event_input_action	input_action	Event input actions

7.6.2.3. Struct dma_resource

Structure for DMA transfer resource.

Table 7-3. Members

Type	Name	Description
dma_callback_t	callback[]	Array of callback functions for DMA transfer job
uint8_t	callback_enable	Bit mask for enabled callbacks
uint8_t	channel_id	Allocated DMA channel ID
DmacDescriptor *	descriptor	DMA transfer descriptor
enum status_code	job_status	Status of the last job
uint32_t	transferred_size	Transferred data size

7.6.2.4. Struct dma_resource_config

DMA configurations for transfer.

Table 7-4. Members

Type	Name	Description
struct dma_events_config	event_config	DMA events configurations
uint8_t	peripheral_trigger	DMA peripheral trigger index
enum dma_priority_level	priority	DMA transfer priority

Type	Name	Description
bool	run_in_standby	Keep DMA channel enabled in standby sleep mode if true
enum dma_transfer_trigger_action	trigger_action	DMA trigger action

7.6.3. Macro Definitions

7.6.3.1. Macro DMA_INVALID_CHANNEL

```
#define DMA_INVALID_CHANNEL
```

DMA invalid channel number.

7.6.3.2. Macro FEATURE_DMA_CHANNEL_STANDBY

```
#define FEATURE_DMA_CHANNEL_STANDBY
```

7.6.4. Function Definitions

7.6.4.1. Function dma_abort_job()

Abort a DMA transfer.

```
void dma_abort_job(
    struct dma_resource * resource)
```

This function will abort a DMA transfer. The DMA channel used for the DMA resource will be disabled. The block transfer count will also be calculated and written to the DMA resource structure.

Note: The DMA resource will not be freed after calling this function. The function [dma_free\(\)](#) can be used to free an allocated resource.

Table 7-5. Parameters

Data direction	Parameter name	Description
[in, out]	resource	Pointer to the DMA resource

7.6.4.2. Function dma_add_descriptor()

Add a DMA transfer descriptor to a DMA resource.

```
enum status_code dma_add_descriptor(
    struct dma_resource * resource,
    DmacDescriptor * descriptor)
```

This function will add a DMA transfer descriptor to a DMA resource. If there was a transfer descriptor already allocated to the DMA resource, the descriptor will be linked to the next descriptor address.

Table 7-6. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource
[in]	descriptor	Pointer to the transfer descriptor

Table 7-7. Return Values

Return value	Description
STATUS_OK	The descriptor is added to the DMA resource
STATUS_BUSY	The DMA resource was busy and the descriptor is not added

7.6.4.3. Function dma_allocate()

Allocate a DMA with configurations.

```
enum status_code dma_allocate(
    struct dma_resource * resource,
    struct dma_resource_config * config)
```

This function will allocate a proper channel for a DMA transfer request.

Table 7-8. Parameters

Data direction	Parameter name	Description
[in, out]	dma_resource	Pointer to a DMA resource instance
[in]	transfer_config	Configurations of the DMA transfer

Returns

Status of the allocation procedure.

Table 7-9. Return Values

Return value	Description
STATUS_OK	The DMA resource was allocated successfully
STATUS_ERR_NOT_FOUND	DMA resource allocation failed

7.6.4.4. Function dma_descriptor_create()

Create a DMA transfer descriptor with configurations.

```
void dma_descriptor_create(
    DmacDescriptor * descriptor,
    struct dma_descriptor_config * config)
```

This function will set the transfer configurations to the DMA transfer descriptor.

Table 7-10. Parameters

Data direction	Parameter name	Description
[in]	descriptor	Pointer to the DMA transfer descriptor
[in]	config	Pointer to the descriptor configuration structure

7.6.4.5. Function dma_descriptor_get_config_defaults()

Initializes DMA transfer configuration with predefined default values.

```
void dma_descriptor_get_config_defaults(  
    struct dma_descriptor_config * config)
```

This function will initialize a given DMA descriptor configuration structure to a set of known default values. This function should be called on any new instance of the configuration structure before being modified by the user application.

The default configuration is as follows:

- Set the descriptor as valid
- Disable event output
- No block action
- Set beat size as byte
- Enable source increment
- Enable destination increment
- Step size is applied to the destination address
- Address increment is beat size multiplied by 1
- Default transfer size is set to 0
- Default source address is set to NULL
- Default destination address is set to NULL
- Default next descriptor not available

Table 7-11. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to the configuration

7.6.4.6. Function dma_disable_callback()

Disable a callback function for a dedicated DMA resource.

```
void dma_disable_callback(  
    struct dma_resource * resource,  
    enum dma_callback_type type)
```

Table 7-12. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource
[in]	type	Callback function type

7.6.4.7. Function dma_enable_callback()

Enable a callback function for a dedicated DMA resource.

```
void dma_enable_callback(  
    struct dma_resource * resource,  
    enum dma_callback_type type)
```

Table 7-13. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource
[in]	type	Callback function type

7.6.4.8. Function dma_free()

Free an allocated DMA resource.

```
enum status_code dma_free(
    struct dma_resource * resource)
```

This function will free an allocated DMA resource.

Table 7-14. Parameters

Data direction	Parameter name	Description
[in, out]	resource	Pointer to the DMA resource

Returns

Status of the free procedure.

Table 7-15. Return Values

Return value	Description
STATUS_OK	The DMA resource was freed successfully
STATUS_BUSY	The DMA resource was busy and can't be freed
STATUS_ERR_NOT_INITIALIZED	DMA resource was not initialized

7.6.4.9. Function dma_get_config_defaults()

Initializes config with predefined default values.

```
void dma_get_config_defaults(
    struct dma_resource_config * config)
```

This function will initialize a given DMA configuration structure to a set of known default values. This function should be called on any new instance of the configuration structure before being modified by the user application.

The default configuration is as follows:

- Software trigger is used as the transfer trigger
- Priority level 0
- Only software/event trigger
- Requires a trigger for each transaction
- No event input /output
- DMA channel is disabled during sleep mode (if has the feature)

Table 7-16. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to the configuration

7.6.4.10. Function dma_get_job_status()

Get DMA resource status.

```
enum status_code dma_get_job_status(
    struct dma_resource * resource)
```

Table 7-17. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource

Returns

Status of the DMA resource.

7.6.4.11. Function dma_is_busy()

Check if the given DMA resource is busy.

```
bool dma_is_busy(
    struct dma_resource * resource)
```

Table 7-18. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource

Returns

Status which indicates whether the DMA resource is busy.

Table 7-19. Return Values

Return value	Description
true	The DMA resource has an on-going transfer
false	The DMA resource is not busy

7.6.4.12. Function dma_register_callback()

Register a callback function for a dedicated DMA resource.

```
void dma_register_callback(
    struct dma_resource * resource,
    dma_callback_t callback,
    enum dma_callback_type type)
```

There are three types of callback functions, which can be registered:

- Callback for transfer complete
- Callback for transfer error

- Callback for channel suspend

Table 7-20. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource
[in]	callback	Pointer to the callback function
[in]	type	Callback function type

7.6.4.13. Function dma_reset_descriptor()

Reset DMA descriptor.

```
void dma_reset_descriptor(
    struct dma_resource * resource)
```

This function will clear the DESCADDR register of an allocated DMA resource.

7.6.4.14. Function dma_resume_job()

Resume a suspended DMA transfer.

```
void dma_resume_job(
    struct dma_resource * resource)
```

This function try to resume a suspended transfer of a DMA resource.

Table 7-21. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource

7.6.4.15. Function dma_start_transfer_job()

Start a DMA transfer.

```
enum status_code dma_start_transfer_job(
    struct dma_resource * resource)
```

This function will start a DMA transfer through an allocated DMA resource.

Table 7-22. Parameters

Data direction	Parameter name	Description
[in, out]	resource	Pointer to the DMA resource

Returns

Status of the transfer start procedure.

Table 7-23. Return Values

Return value	Description
STATUS_OK	The transfer was started successfully
STATUS_BUSY	The DMA resource was busy and the transfer was not started
STATUS_ERR_INVALID_ARG	Transfer size is 0 and transfer was not started

7.6.4.16. Function dma_suspend_job()

Suspend a DMA transfer.

```
void dma_suspend_job(
    struct dma_resource * resource)
```

This function will request to suspend the transfer of the DMA resource. The channel is kept enabled, can receive transfer triggers (the transfer pending bit will be set), but will be removed from the arbitration scheme. The channel operation can be resumed by calling [dma_resume_job\(\)](#).

Note: This function sets the command to suspend the DMA channel associated with a DMA resource. The channel suspend interrupt flag indicates whether the transfer is truly suspended.

Table 7-24. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource

7.6.4.17. Function dma_trigger_transfer()

Will set a software trigger for resource.

```
void dma_trigger_transfer(
    struct dma_resource * resource)
```

This function is used to set a software trigger on the DMA channel associated with resource. If a trigger is already pending no new trigger will be generated for the channel.

Table 7-25. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource

7.6.4.18. Function dma_unregister_callback()

Unregister a callback function for a dedicated DMA resource.

```
void dma_unregister_callback(
    struct dma_resource * resource,
    enum dma_callback_type type)
```

There are three types of callback functions:

- Callback for transfer complete
- Callback for transfer error
- Callback for channel suspend

The application can unregister any of the callback functions which are already registered and are no longer needed.

Table 7-26. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to the DMA resource
[in]	type	Callback function type

7.6.4.19. Function dma_update_descriptor()

Update DMA descriptor.

```
void dma_update_descriptor(
    struct dma_resource * resource,
    DmacDescriptor * descriptor)
```

This function can update the descriptor of an allocated DMA resource.

7.6.5. Enumeration Definitions

7.6.5.1. Enum dma_address_increment_stepsize

Address increment step size. These bits select the address increment step size. The setting apply to source or destination address, depending on STEPSEL setting.

Table 7-27. Members

Enum value	Description
DMA_ADDRESS_INCREMENT_STEP_SIZE_1	The address is incremented by (beat size * 1).
DMA_ADDRESS_INCREMENT_STEP_SIZE_2	The address is incremented by (beat size * 2).
DMA_ADDRESS_INCREMENT_STEP_SIZE_4	The address is incremented by (beat size * 4).
DMA_ADDRESS_INCREMENT_STEP_SIZE_8	The address is incremented by (beat size * 8).
DMA_ADDRESS_INCREMENT_STEP_SIZE_16	The address is incremented by (beat size * 16).
DMA_ADDRESS_INCREMENT_STEP_SIZE_32	The address is incremented by (beat size * 32).
DMA_ADDRESS_INCREMENT_STEP_SIZE_64	The address is incremented by (beat size * 64).
DMA_ADDRESS_INCREMENT_STEP_SIZE_128	The address is incremented by (beat size * 128).

7.6.5.2. Enum dma_beat_size

The basic transfer unit in DMAC is a beat, which is defined as a single bus access. Its size is configurable and applies to both read and write.

Table 7-28. Members

Enum value	Description
DMA_BEAT_SIZE_BYTE	8-bit access.
DMA_BEAT_SIZE_HWORD	16-bit access.
DMA_BEAT_SIZE_WORD	32-bit access.

7.6.5.3. Enum dma_block_action

Block action definitions.

Table 7-29. Members

Enum value	Description
DMA_BLOCK_ACTION_NOACT	No action.
DMA_BLOCK_ACTION_INT	Channel in normal operation and sets transfer complete interrupt flag after block transfer.
DMA_BLOCK_ACTION_SUSPEND	Trigger channel suspend after block transfer and sets channel suspend interrupt flag once the channel is suspended.
DMA_BLOCK_ACTION_BOTH	Sets transfer complete interrupt flag after a block transfer and trigger channel suspend. The channel suspend interrupt flag will be set once the channel is suspended.

7.6.5.4. Enum dma_callback_type

Callback types for DMA callback driver.

Table 7-30. Members

Enum value	Description
DMA_CALLBACK_TRANSFER_ERROR	Callback for any of transfer errors. A transfer error is flagged if a bus error is detected during an AHB access or when the DMAC fetches an invalid descriptor.
DMA_CALLBACK_TRANSFER_DONE	Callback for transfer complete.
DMA_CALLBACK_CHANNEL_SUSPEND	Callback for channel suspend.
DMA_CALLBACK_N	Number of available callbacks.

7.6.5.5. Enum dma_event_input_action

DMA input actions.

Table 7-31. Members

Enum value	Description
DMA_EVENT_INPUT_NOACT	No action.
DMA_EVENT_INPUT_TRIG	Normal transfer and periodic transfer trigger.

Enum value	Description
DMA_EVENT_INPUT_CTRIG	Conditional transfer trigger.
DMA_EVENT_INPUT_CBLOCK	Conditional block transfer.
DMA_EVENT_INPUT_SUSPEND	Channel suspend operation.
DMA_EVENT_INPUT_RESUME	Channel resume operation.
DMA_EVENT_INPUT_SSKIP	Skip next block suspend action.

7.6.5.6. Enum dma_event_output_selection

Event output selection.

Table 7-32. Members

Enum value	Description
DMA_EVENT_OUTPUT_DISABLE	Event generation disable.
DMA_EVENT_OUTPUT_BLOCK	Event strobe when block transfer complete.
DMA_EVENT_OUTPUT_RESERVED	Event output reserved.
DMA_EVENT_OUTPUT_BEAT	Event strobe when beat transfer complete.

7.6.5.7. Enum dma_priority_level

DMA priority level.

Table 7-33. Members

Enum value	Description
DMA_PRIORITY_LEVEL_0	Priority level 0.
DMA_PRIORITY_LEVEL_1	Priority level 1.
DMA_PRIORITY_LEVEL_2	Priority level 2.
DMA_PRIORITY_LEVEL_3	Priority level 3.

7.6.5.8. Enum dma_step_selection

DMA step selection. This bit determines whether the step size setting is applied to source or destination address.

Table 7-34. Members

Enum value	Description
DMA_STEPSEL_DST	Step size settings apply to the destination address.
DMA_STEPSEL_SRC	Step size settings apply to the source address.

7.6.5.9. Enum dma_transfer_trigger_action

DMA trigger action type.

Table 7-35. Members

Enum value	Description
DMA_TRIGGER_ACTION_BLOCK	Perform a block transfer when triggered.
DMA_TRIGGER_ACTION_BEAT	Perform a beat transfer when triggered.
DMA_TRIGGER_ACTION_TRANSACTION	Perform a transaction when triggered.

7.7. Extra Information for DMAC Driver

7.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
CPU	Central Processing Unit

7.7.2. Dependencies

This driver has the following dependencies:

- System Clock Driver

7.7.3. Errata

There are no errata related to this driver.

7.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Add SAM C21 support
Add SAM L21 support
Initial Release

7.8. Examples for DMAC Driver

This is a list of the available Quick Start Guides (QSGs) and example applications for [SAM Direct Memory Access Controller \(DMAC\) Driver](#). QSGs are simple examples with step-by-step instructions to

configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for Memory to Memory Data Transfer Using DMAC](#)

Note: More DMA usage examples are available in peripheral QSGs. A quick start guide for TC/TCC shows the usage of DMA event trigger; SERCOM SPI/USART/I²C has example for DMA transfer from peripheral to memory or from memory to peripheral; ADC/DAC shows peripheral to peripheral transfer.

7.8.1. Quick Start Guide for Memory to Memory Data Transfer Using DMAC

The supported board list:

- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM D11 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM DA1 Xplained Pro

In this use case, the DMAC is configured for:

- Moving data from memory to memory
- Using software trigger
- Using DMA priority level 0
- Transaction as DMA trigger action
- No action on input events
- Output event not enabled

7.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
#define DATA_LENGTH (512)

static uint8_t source_memory[DATA_LENGTH];
static uint8_t destination_memory[DATA_LENGTH];
static volatile bool transfer_is_done = false;

COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor SECTION_DMACH_DESCRIPTOR;

static void transfer_done(struct dma_resource* const resource )
{
    transfer_is_done = true;
}

static void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    dma_allocate(resource, &config);
```

```

}

static void setup_transfer_descriptor(DmacDescriptor *descriptor )
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = sizeof(source_memory);
    descriptor_config.source_address = (uint32_t)source_memory +
        sizeof(source_memory);
    descriptor_config.destination_address = (uint32_t)destination_memory +
        sizeof(source_memory);

    dma_descriptor_create(descriptor, &descriptor_config);
}

```

Add the below section to user application initialization (typically the start of `main()`):

```

configure_dma_resource(&example_resource);
setup_transfer_descriptor(&example_descriptor);
dma_add_descriptor(&example_resource, &example_descriptor);
dma_register_callback(&example_resource, transfer_done,
    DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource, DMA_CALLBACK_TRANSFER_DONE);
for (uint32_t i = 0; i < DATA_LENGTH; i++) {
    source_memory[i] = i;
}

```

Workflow

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

2. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

4. Declare a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

5. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

6. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address. In this example, we have enabled the source and destination address increment. The source and destination addresses to be stored into `descriptor_config` must correspond to the end of the transfer.

```
descriptor_config.block_transfer_count = sizeof(source_memory);
descriptor_config.source_address = (uint32_t)source_memory +
    sizeof(source_memory);
descriptor_config.destination_address = (uint32_t)destination_memory +
    sizeof(source_memory);
```

7. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

8. Add the DMA transfer descriptor to the allocated DMA resource.

```
dma_add_descriptor(&example_resource, &example_descriptor);
```

9. Register a callback to indicate transfer status.

```
dma_register_callback(&example_resource, transfer_done,
    DMA_CALLBACK_TRANSFER_DONE);
```

10. Set the transfer done flag in the registered callback function.

```
static void transfer_done(struct dma_resource* const resource )
{
    transfer_is_done = true;
}
```

11. Enable the registered callbacks.

```
dma_enable_callback(&example_resource, DMA_CALLBACK_TRANSFER_DONE);
```

7.8.1.2. Use Case

Code

Add the following code at the start of `main()`:

```
struct dma_resource example_resource;
```

Copy the following code to your user application:

```
dma_start_transfer_job(&example_resource);
dma_trigger_transfer(&example_resource);
while (!transfer_is_done) {
    /* Wait for transfer done */
}

while (true) {
    /* Nothing to do */
}
```

Workflow

1. Start the DMA transfer job with the allocated DMA resource and transfer descriptor.

```
dma_start_transfer_job(&example_resource);
```

2. Set the software trigger for the DMA channel. This can be done before or after the DMA job is started. Note that all transfers needs a trigger to start.

```
dma_trigger_transfer(&example_resource);
```

3. Waiting for the setting of the transfer done flag.

```
while (!transfer_is_done) {  
    /* Wait for transfer done */  
}
```

8. SAM Event System (EVENTS) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's peripheral event resources and users within the device, including enabling and disabling of peripheral source selection and synchronization of clock domains between various modules. The following API modes is covered by this manual:

- Polled API
- Interrupt hook API

The following peripheral is used by this module:

- EVSYS (Event System Management)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

8.1. Prerequisites

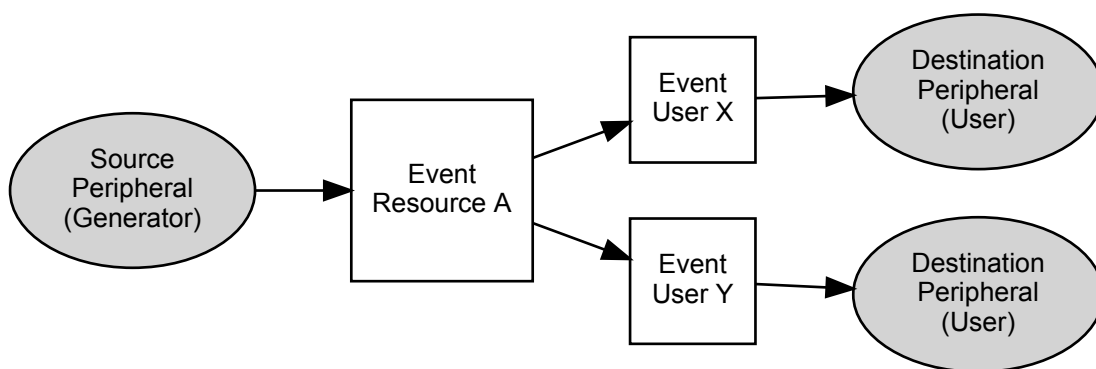
There are no prerequisites for this module.

8.2. Module Overview

Peripherals within the SAM devices are capable of generating two types of actions in response to given stimulus; set a register flag for later intervention by the CPU (using interrupt or polling methods), or generate event signals, which can be internally routed directly to other peripherals within the device. The use of events allows for direct actions to be performed in one peripheral in response to a stimulus in another without CPU intervention. This can lower the overall power consumption of the system if the CPU is able to remain in sleep modes for longer periods (SleepWalking), and lowers the latency of the system response.

The event system is comprised of a number of freely configurable Event resources, plus a number of fixed Event Users. Each Event resource can be configured to select the input peripheral that will generate the events signal, as well as the synchronization path and edge detection mode. The fixed-function Event Users, connected to peripherals within the device, can then subscribe to an Event resource in a one-to-many relationship in order to receive events as they are generated. An overview of the event system chain is shown in [Figure 8-1](#).

Figure 8-1. Module Overview



There are many different events that can be routed in the device, which can then trigger many different actions. For example, an Analog Comparator module could be configured to generate an event when the input signal rises above the compare threshold, which then triggers a Timer Counter module to capture the current count value for later use.

8.2.1. Event Channels

The Event module in each device consists of several channels, which can be freely linked to an event generator (i.e. a peripheral within the device that is capable of generating events). Each channel can be individually configured to select the generator peripheral, signal path, and edge detection applied to the input event signal, before being passed to any event user(s).

Event channels can support multiple users within the device in a standardized manner. When an Event User is linked to an Event Channel, the channel will automatically handshake with all attached users to ensure that all modules correctly receive and acknowledge the event.

8.2.2. Event Users

Event Users are able to subscribe to an Event Channel, once it has been configured. Each Event User consists of a fixed connection to one of the peripherals within the device (for example, an ADC module, or Timer module) and is capable of being connected to a single Event Channel.

8.2.3. Edge Detection

For asynchronous events, edge detection on the event input is not possible, and the event signal must be passed directly between the event generator and event user. For synchronous and re-synchronous events, the input signal from the event generator must pass through an edge detection unit, so that only the rising, falling, or both edges of the event signal triggers an action in the event user.

8.2.4. Path Selection

The event system in the SAM devices supports three signal path types from the event generator to Event Users: asynchronous, synchronous, and re-synchronous events.

8.2.4.1. Asynchronous Paths

Asynchronous event paths allow for an asynchronous connection between the event generator and Event Users, when the source and destination peripherals share the same Generic Clock channel. In this mode the event is propagated between the source and destination directly to reduce the event latency, thus no edge detection is possible. The asynchronous event chain is shown in [Figure 8-2](#).

Figure 8-2. Asynchronous Paths



Note: Identically shaped borders in the diagram indicate a shared generic clock channel.

8.2.4.2. Synchronous Paths

The Synchronous event path should be used when edge detection or interrupts from the event channel are required, and the source event generator and the event channel shares the same Generic Clock channel. The synchronous event chain is shown in [Figure 8-3](#).

Not all peripherals support Synchronous event paths; refer to the device datasheet.

Figure 8-3. Synchronous Paths



Note: Identically shaped borders in the diagram indicate a shared generic clock channel.

8.2.4.3. Re-synchronous Paths

Re-synchronous event paths are a special form of synchronous events, where when edge detection or interrupts from the event channel are required, but the event generator and the event channel use different Generic Clock channels. The re-synchronous path allows the Event System to synchronize the incoming event signal from the Event Generator to the clock of the Event System module to avoid missed events, at the cost of a higher latency due to the re-synchronization process. The re-synchronous event chain is shown in [Figure 8-4](#).

Not all peripherals support re-synchronous event paths; refer to the device datasheet.

Figure 8-4. Re-synchronous Paths

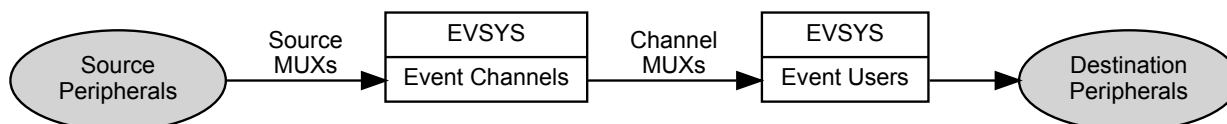


Note: Identically shaped borders in the diagram indicate a shared generic clock channel.

8.2.5. Physical Connection

[Figure 8-5](#) shows how this module is interconnected within the device.

Figure 8-5. Physical Connection



8.2.6. Configuring Events

For SAM devices, several steps are required to properly configure an event chain, so that hardware peripherals can respond to events generated by each other, as listed below.

8.2.6.1. Source Peripheral

1. The source peripheral (that will generate events) must be configured and enabled.
2. The source peripheral (that will generate events) must have an output event enabled.

8.2.6.2. Event System

1. An event system channel must be allocated and configured with the correct source peripheral selected as the channel's event generator.
2. The event system user must be configured and enabled, and attached to # event channel previously allocated.

8.2.6.3. Destination Peripheral

1. The destination peripheral (that will receive events) must be configured and enabled.
2. The destination peripheral (that will receive events) must have an input event enabled.

8.3. Special Considerations

There are no special considerations for this module.

8.4. Extra Information

For extra information, see [Extra Information for EVENTS Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

8.5. Examples

For a list of examples related to this driver, see [Examples for EVENTS Driver](#).

8.6. API Overview

8.6.1. Variable and Type Definitions

8.6.1.1. Type events_interrupt_hook

```
typedef void(* events_interrupt_hook )(struct events_resource *resource)
```

8.6.2. Structure Definitions

8.6.2.1. Struct events_config

This event configuration struct is used to configure each of the channels.

Table 8-1. Members

Type	Name	Description
uint8_t	clock_source	Clock source for the event channel
enum events_edge_detect	edge_detect	Select edge detection mode
uint8_t	generator	Set event generator for the channel
enum events_path_selection	path	Select events channel path

8.6.2.2. Struct events_hook

Event hook structure.

Table 8-2. Members

Type	Name	Description
events_interrupt_hook	hook_func	Event hook function
struct events_hook *	next	Next event hook
struct events_resource *	resource	Event resource

8.6.2.3. Struct events_resource

Event resource structure.

Note: The fields in this structure should not be altered by the user application; they are reserved for driver internals only.

8.6.3. Macro Definitions

8.6.3.1. Macro EVSYS_ID_GEN_NONE

```
#define EVSYS_ID_GEN_NONE
```

Use this to disable any peripheral event input to a channel. This can be useful if you only want to use a channel for software generated events. Definition for no generator selection.

8.6.4. Function Definitions

8.6.4.1. Function events_ack_interrupt()

Acknowledge an interrupt source.

```
enum status_code events_ack_interrupt(
    struct events_resource * resource,
    enum events_interrupt_source source)
```

Acknowledge an interrupt source so the interrupt state is cleared in hardware.

Table 8-3. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns

Status of the interrupt source.

Table 8-4. Return Values

Return value	Description
STATUS_OK	Interrupt source was acknowledged successfully

8.6.4.2. Function events_add_hook()

Insert hook into the event drivers interrupt hook queue.

```
enum status_code events_add_hook(
    struct events_resource * resource,
    struct events_hook * hook)
```

Inserts a hook into the event drivers interrupt hook queue.

Table 8-5. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	hook	Pointer to an events_hook struct instance

Returns

Status of the insertion procedure.

Table 8-6. Return Values

Return value	Description
STATUS_OK	Insertion of hook went successful

8.6.4.3. Function events_allocate()

Allocate an event channel and set configuration.

```
enum status_code events_allocate(
    struct events_resource * resource,
    struct events_config * config)
```

Allocates an event channel from the event channel pool and sets the channel configuration.

Table 8-7. Parameters

Data direction	Parameter name	Description
[out]	resource	Pointer to a events_resource struct instance
[in]	config	Pointer to a events_config struct

Returns

Status of the configuration procedure.

Table 8-8. Return Values

Return value	Description
STATUS_OK	Allocation and configuration went successful
STATUS_ERR_NOT_FOUND	No free event channel found

8.6.4.4. Function events_attach_user()

Attach user to the event channel.

```
enum status_code events_attach_user(
    struct events_resource * resource,
    uint8_t user_id)
```

Attach a user peripheral to the event channel to receive events.

Table 8-9. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	user_id	A number identifying the user peripheral found in the device header file

Returns

Status of the user attach procedure.

Table 8-10. Return Values

Return value	Description
STATUS_OK	No errors detected when attaching the event user

8.6.4.5. Function events_create_hook()

Initializes an interrupt hook for insertion in the event interrupt hook queue.

```
enum status_code events_create_hook(
    struct events_hook * hook,
    events_interrupt_hook hook_func)
```

Initializes a hook structure so it is ready for insertion in the interrupt hook queue.

Table 8-11. Parameters

Data direction	Parameter name	Description
[out]	hook	Pointer to an events_hook struct instance
[in]	hook_func	Pointer to a function containing the interrupt hook code

Returns

Status of the hook creation procedure.

Table 8-12. Return Values

Return value	Description
STATUS_OK	Creation and initialization of interrupt hook went successful

8.6.4.6. Function events_del_hook()

Remove hook from the event drivers interrupt hook queue.

```
enum status_code events_del_hook(
    struct events_resource * resource,
    struct events_hook * hook)
```

Removes a hook from the event drivers interrupt hook queue.

Table 8-13. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	hook	Pointer to an events_hook struct instance

Returns

Status of the removal procedure.

Table 8-14. Return Values

Return value	Description
STATUS_OK	Removal of hook went successful
STATUS_ERR_NO_MEMORY	There are no hooks instances in the event driver interrupt hook list
STATUS_ERR_NOT_FOUND	Interrupt hook not found in the event drivers interrupt hook list

8.6.4.7. Function events_detach_user()

Detach a user peripheral from the event channel.

```
enum status_code events_detach_user(
    struct events_resource * resource,
    uint8_t user_id)
```

Deattach a user peripheral from the event channels so it does not receive any more events.

Table 8-15. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an event_resource struct instance
[in]	user_id	A number identifying the user peripheral found in the device header file

Returns

Status of the user detach procedure.

Table 8-16. Return Values

Return value	Description
STATUS_OK	No errors detected when detaching the event user

8.6.4.8. Function events_disable_interrupt_source()

Disable interrupt source.

```
enum status_code events_disable_interrupt_source(
    struct events_resource * resource,
    enum events_interrupt_source source)
```

Disable an interrupt source so can trigger execution of an interrupt hook.

Table 8-17. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns

Status of the interrupt source enable procedure.

Table 8-18. Return Values

Return value	Description
STATUS_OK	Enabling of the interrupt source went successful
STATUS_ERR_INVALID_ARG	Interrupt source does not exist

8.6.4.9. Function events_enable_interrupt_source()

Enable interrupt source.

```
enum status_code events_enable_interrupt_source(
    struct events_resource * resource,
    enum events_interrupt_source source)
```

Enable an interrupt source so can trigger execution of an interrupt hook.

Table 8-19. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns

Status of the interrupt source enable procedure.

Table 8-20. Return Values

Return value	Description
STATUS_OK	Enabling of the interrupt source was successful
STATUS_ERR_INVALID_ARG	Interrupt source does not exist

8.6.4.10. Function `events_get_config_defaults()`

Initializes an event configurations struct to defaults.

```
void events_get_config_defaults(
    struct events_config * config)
```

Initializes an event configuration struct to predefined safe default settings.

Table 8-21. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an instance of struct events_config

8.6.4.11. Function `events_get_free_channels()`

Get the number of free channels.

```
uint8_t events_get_free_channels( void )
```

Get the number of allocatable channels in the events system resource pool.

Returns

The number of free channels in the event system.

8.6.4.12. Function `events_is_busy()`

Check if a channel is busy.

```
bool events_is_busy(
    struct events_resource * resource)
```

Check if a channel is busy, a channel stays busy until all users connected to the channel has handled an event.

Table 8-22. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to a events_resource struct instance

Returns

Status of the channels busy state.

Table 8-23. Return Values

Return value	Description
true	One or more users connected to the channel has not handled the last event
false	All users are ready to handle new events

8.6.4.13. Function events_is_detected()

Check if an event is detected on the event channel.

```
bool events_is_detected(
    struct events_resource * resource)
```

Check if an event has been detected on the channel.

Note: This function will clear the event detected interrupt flag.

Table 8-24. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns

Status of the event detection interrupt flag.

Table 8-25. Return Values

Return value	Description
true	Event has been detected
false	Event has not been detected

8.6.4.14. Function events_is_interrupt_set()

Check if interrupt source is set.

```
bool events_is_interrupt_set(
    struct events_resource * resource,
    enum events_interrupt_source source)
```

Check if an interrupt source is set and should be processed.

Table 8-26. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns

Status of the interrupt source.

Table 8-27. Return Values

Return value	Description
true	Interrupt source is set
false	Interrupt source is not set

8.6.4.15. Function events_is_overrun()

Check if there has been an overrun situation on this channel.

```
bool events_is_overrun(
    struct events_resource * resource)
```

Note: This function will clear the event overrun detected interrupt flag.

Table 8-28. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns

Status of the event overrun interrupt flag.

Table 8-29. Return Values

Return value	Description
true	Event overrun has been detected
false	Event overrun has not been detected

8.6.4.16. Function events_is_users_ready()

Check if all users connected to the channel are ready.

```
bool events_is_users_ready(
    struct events_resource * resource)
```

Check if all users connected to the channel are ready to handle incoming events.

Table 8-30. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns

The ready status of users connected to an event channel.

Table 8-31. Return Values

Return value	Description
true	All the users connected to the event channel are ready to handle incoming events
false	One or more users connected to the event channel are not ready to handle incoming events

8.6.4.17. Function `events_release()`

Release allocated channel back the the resource pool.

```
enum status_code events_release(  
    struct events_resource * resource)
```

Release an allocated channel back to the resource pool to make it available for other purposes.

Table 8-32. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an <code>events_resource</code> struct

Returns

Status of the channel release procedure.

Table 8-33. Return Values

Return value	Description
STATUS_OK	No error was detected when the channel was released
STATUS_BUSY	One or more event users have not processed the last event
STATUS_ERR_NOT_INITIALIZED	Channel not allocated, and can therefore not be released

8.6.4.18. Function `events_trigger()`

Trigger software event.

```
enum status_code events_trigger(  
    struct events_resource * resource)
```

Trigger an event by software.

Note: Software event works on either a synchronous path or resynchronized path, and edge detection must be configured to rising-edge detection.

Table 8-34. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an <code>events_resource</code> struct

Returns

Status of the event software procedure.

Table 8-35. Return Values

Return value	Description
STATUS_OK	No error was detected when the software trigger signal was issued
STATUS_ERR_UNSUPPORTED_DEV	If the channel path is asynchronous and/or the edge detection is not set to RISING

8.6.5. Enumeration Definitions

8.6.5.1. Enum events_edge_detect

Event channel edge detect setting.

Table 8-36. Members

Enum value	Description
EVENTS_EDGE_DETECT_NONE	No event output
EVENTS_EDGE_DETECT_RISING	Event on rising edge
EVENTS_EDGE_DETECT_FALLING	Event on falling edge
EVENTS_EDGE_DETECT_BOTH	Event on both edges

8.6.5.2. Enum events_interrupt_source

Interrupt source selector definitions.

Table 8-37. Members

Enum value	Description
EVENTS_INTERRUPT_OVERRUN	Overflow in event channel detected interrupt
EVENTS_INTERRUPT_DETECT	Event signal propagation in event channel detected interrupt

8.6.5.3. Enum events_path_selection

Event channel path selection.

Table 8-38. Members

Enum value	Description
EVENTS_PATH_SYNCHRONOUS	Select the synchronous path for this event channel
EVENTS_PATH_RESYNCHRONIZED	Select the resynchronizer path for this event channel
EVENTS_PATH_ASYNCHRONOUS	Select the asynchronous path for this event channel

8.7. Extra Information for EVENTS Driver

8.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
CPU	Central Processing Unit
MUX	Multiplexer

8.7.2. Dependencies

This driver has the following dependencies:

- System Clock Driver

8.7.3. Errata

There are no errata related to this driver.

8.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Fix a bug in internal function <code>_events_find_bit_position()</code>
Rewrite of events driver
Initial Release

8.8. Examples for EVENTS Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Event System \(EVENTS\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for EVENTS - Basic](#)
- [Quick Start Guide for EVENTS - Interrupt Hooks](#)

8.8.1. Quick Start Guide for EVENTS - Basic

In this use case, the EVENT module is configured for:

- Synchronous event path with rising edge detection on the input
- One user attached to the configured event channel
- No hardware event generator attached to the channel

This use case allocates an event channel. This channel is not connected to any hardware event generator, events are software triggered. One user is connected to the allocated and configured event channel.

8.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D20 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER        EVSYS_ID_USER_TC3_EVU
```

- SAM D21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER        EVSYS_ID_USER_TC3_EVU
```

- SAM R21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER        EVSYS_ID_USER_TC3_EVU
```

- SAM D11 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC2_MCX_0
#define CONF_EVENT_USER        EVSYS_ID_USER_TC1_EVU
```

- SAM L21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_NONE
#define CONF_EVENT_USER        EVSYS_ID_USER_PORT_EV_0
```

- SAM L22 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_NONE
#define CONF_EVENT_USER        EVSYS_ID_USER_PORT_EV_0
```

- SAM DA1 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER        EVSYS_ID_USER_TC3_EVU
```

- SAM C21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_NONE
#define CONF_EVENT_USER        EVSYS_ID_USER_PORT_EV_0
```

Copy-paste the following setup code to your user application:

```
static void configure_event_channel(struct events_resource *resource)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator      = CONF_EVENT_GENERATOR;
    config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
    config.path           = EVENTS_PATH_SYNCHRONOUS;
    config.clock_source   = GCLK_GENERATOR_0;
```

```

    events_allocate(resource, &config);
}

static void configure_event_user(struct events_resource *resource)
{
    events_attach_user(resource, CONF_EVENT_USER);
}

```

Create an event resource struct and add to user application (typically the start of `main()`):

```
struct events_resource example_event;
```

Add to user application initialization (typically the start of `main()`):

```
configure_event_channel(&example_event);
configure_event_user(&example_event);
```

Workflow

1. Create an event channel configuration struct, which can be filled out to adjust the configuration of a single event channel.

```
struct events_config config;
```

2. Initialize the event channel configuration struct with the module's default values.

```
events_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to request that the channel is to be attached to the specified event generator, that rising edges of the event signal is to be detected on the channel, and that the synchronous event path is to be used.

```

config.generator      = CONF_EVENT_GENERATOR;
config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
config.path           = EVENTS_PATH_SYNCHRONOUS;
config.clock_source   = GCLK_GENERATOR_0;

```

4. Allocate and configure the channel using the configuration structure.

```
events_allocate(resource, &config);
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

5. Attach a user to the channel.

```
events_attach_user(resource, CONF_EVENT_USER);
```

8.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```

while (events_is_busy(&example_event)) {
    /* Wait for channel */
};

events_trigger(&example_event);

while (true) {

```

```

    /* Nothing to do */
}

```

Workflow

1. Wait for the event channel to become ready to accept a new event trigger.

```

while (events_is_busy(&example_event)) {
    /* Wait for channel */
};

```

2. Perform a software event trigger on the configured event channel.

```

events_trigger(&example_event);

```

8.8.2. Quick Start Guide for EVENTS - Interrupt Hooks

In this use case, the EVENT module is configured for:

- Synchronous event path with rising edge detection
- TC4 as event generator on the allocated event channel (TC0 is used for SAM L22)
- One event channel user attached
- An event interrupt hook is used to execute some code when an event is detected

In this use case TC is used as event generator, generating events on overflow. One user attached, counting events on the channel. To be able to execute some code when an event is detected, an interrupt hook is used. The interrupt hook will also count the number of events detected and toggle a LED on the board each time an event is detected.

Note: Because this example is showing how to set up an interrupt hook there is no user attached to the user.

8.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use case.

Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D20 Xplained Pro:

```

#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_TC0_EVU

```

```

#define CONF_TC_MODULE TC4

```

- SAM D21 Xplained Pro:

```

#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DMAC_CH_0

```

```

#define CONF_TC_MODULE TC4

```


- **SAM R21 Xplained Pro:**

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DM̄AC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- **SAM D11 Xplained Pro:**

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC2_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DM̄AC_CH_0
```

```
#define CONF_TC_MODULE TC2
```

- **SAM L21 Xplained Pro:**

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DM̄AC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- **SAM L22 Xplained Pro:**

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC0_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DM̄AC_CH_0
```

```
#define CONF_TC_MODULE TC0
```

- **SAM DA1 Xplained Pro:**

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DM̄AC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- **SAM C21 Xplained Pro:**

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER        EVSYS_ID_USER_DM̄AC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

Copy-paste the following setup code to your user application:

```
static volatile uint32_t event_count = 0;

void event_counter(struct events_resource *resource);

static void configure_event_channel(struct events_resource *resource)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator      = CONF_EVENT_GENERATOR;
    config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
    config.path           = EVENTS_PATH_SYNCHRONOUS;
    config.clock_source   = GCLK_GENERATOR_0;

    events_allocate(resource, &config);
}
```

```

static void configure_event_user(struct events_resource *resource)
{
    events_attach_user(resource, CONF_EVENT_USER);
}

static void configure_tc(struct tc_module *tc_instance)
{
    struct tc_config config_tc;
    struct tc_events config_events;

    tc_get_config_defaults(&config_tc);

    config_tc.counter_size      = TC_COUNTER_SIZE_8BIT;
    config_tc.wave_generation   = TC_WAVE_GENERATION_NORMAL_FREQ;
    config_tc.clock_source      = GCLK_GENERATOR_1;
    config_tc.clock_prescaler   = TC_CLOCK_PRESCALER_DIV64;

    tc_init(tc_instance, CONF_TC_MODULE, &config_tc);

    config_events.generate_event_on_overflow = true;
    tc_enable_events(tc_instance, &config_events);

    tc_enable(tc_instance);
}

static void configure_event_interrupt(struct events_resource *resource,
                                     struct events_hook *hook)
{
    events_create_hook(hook, event_counter);

    events_add_hook(resource, hook);
    events_enable_interrupt_source(resource, EVENTS_INTERRUPT_DETECT);
}

void event_counter(struct events_resource *resource)
{
    if(events_is_interrupt_set(resource, EVENTS_INTERRUPT_DETECT)) {
        port_pin_toggle_output_level(LED_0_PIN);

        event_count++;
        events_ack_interrupt(resource, EVENTS_INTERRUPT_DETECT);
    }
}

```

Add to user application initialization (typically the start of `main()`):

```

struct tc_module      tc_instance;
struct events_resource example_event;
struct events_hook     hook;

system_init();
system_interrupt_enable_global();

configure_event_channel(&example_event);
configure_event_user(&example_event);
configure_event_interrupt(&example_event, &hook);
configure_tc(&tc_instance);

```

Workflow

1. Create an event channel configuration structure instance which will contain the configuration for the event.

```
struct events_config config;
```

2. Initialize the event channel configuration struct with safe default values.

Note: This shall always be performed before using the configuration struct to ensure that all members are initialized to known default values.

```
events_get_config_defaults(&config);
```

3. Adjust the configuration structure:

- Use EXAMPLE_EVENT_GENRATOR as event generator
- Detect events on rising edge
- Use the synchronous event path
- Use GCLK Generator 0 as event channel clock source

```
config.generator      = CONF_EVENT_GENERATOR;  
config.edge_detect    = EVENTS_EDGE_DETECT_RISING;  
config.path           = EVENTS_PATH_SYNCHRONOUS;  
config.clock_source    = GCLK_GENERATOR_0;
```

4. Allocate and configure the channel using the configuration structure.

```
events_allocate(resource, &config);
```

5. Make sure there is no user attached. To attach a user, change the value of EXAMPLE_EVENT_USER to the correct peripheral ID.

```
events_attach_user(resource, CONF_EVENT_USER);
```

6. Create config_tc and config_events configuration structure instances.

```
struct tc_config config_tc;  
struct tc_events config_events;
```

7. Initialize the TC module configuration structure with safe default values.

Note: This function shall always be called on new configuration structure instances to make sure that all structure members are initialized.

```
tc_get_config_defaults(&config_tc);
```

8. Adjust the config_tc structure:

- Set counter size to 8-bit
- Set wave generation mode to normal frequency generation
- Use GCLK generator 1 to as TC module clock source
- Prescale the input clock with 64

```
config_tc.counter_size    = TC_COUNTER_SIZE_8BIT;  
config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_FREQ;  
config_tc.clock_source    = GCLK_GENERATOR_1;  
config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV64;
```

9. Initialize, configure, and associate the `tc_instance` handle with the TC hardware pointed to by `TC_MODULE`.

```
tc_init(tc_instance, CONF_TC_MODULE, &config_tc);
```

10. Adjust the `config_events` structure to enable event generation on overflow in the timer and then enable the event configuration.

```
config_events.generate_event_on_overflow = true;  
tc_enable_events(tc_instance, &config_events);
```

11. Enable the timer/counter module.

```
tc_enable(tc_instance);
```

12. Create a new interrupt hook and use the function `event_counter` as hook code.

```
events_create_hook(hook, event_counter);
```

13. Add the newly created hook to the interrupt hook queue and enable the event detected interrupt.

```
events_add_hook(resource, hook);  
events_enable_interrupt_source(resource, EVENTS_INTERRUPT_DETECT);
```

14. Example interrupt hook code. If the hook was triggered by an event detected interrupt on the event channel this code will toggle the LED on the Xplained PRO board and increase the value of the `event_count` variable. The interrupt is then acknowledged.

```
void event_counter(struct events_resource *resource)  
{  
    if(events_is_interrupt_set(resource, EVENTS_INTERRUPT_DETECT)) {  
        port_pin_toggle_output_level(LED_0_PIN);  
  
        event_count++;  
        events_ack_interrupt(resource, EVENTS_INTERRUPT_DETECT);  
    }  
}
```

8.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (events_is_busy(&example_event)) {  
    /* Wait for channel */  
};  
  
tc_start_counter(&tc_instance);  
  
while (true) {  
    /* Nothing to do */  
}
```

Workflow

1. Wait for the event channel to become ready.

```
while (events_is_busy(&example_event)) {  
    /* Wait for channel */  
};
```

2. Start the timer/counter.

```
tc_start_counter(&tc_instance);
```

9. SAM External Interrupt (EXTINT) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of external interrupts generated by the physical device pins, including edge detection. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- EIC (External Interrupt Controller)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

9.1. Prerequisites

There are no prerequisites for this module.

9.2. Module Overview

The External Interrupt (EXTINT) module provides a method of asynchronously detecting rising edge, falling edge, or specific level detection on individual I/O pins of a device. This detection can then be used to trigger a software interrupt or event, or polled for later use if required. External interrupts can also optionally be used to automatically wake up the device from sleep mode, allowing the device to conserve power while still being able to react to an external stimulus in a timely manner.

9.2.1. Logical Channels

The External Interrupt module contains a number of logical channels, each of which is capable of being individually configured for a given pin routing, detection mode, and filtering/wake up characteristics.

Each individual logical external interrupt channel may be routed to a single physical device I/O pin in order to detect a particular edge or level of the incoming signal.

9.2.2. NMI Channels

One or more Non Maskable Interrupt (NMI) channels are provided within each physical External Interrupt Controller module, allowing a single physical pin of the device to fire a single NMI interrupt in response to a particular edge or level stimulus. An NMI cannot, as the name suggests, be disabled in firmware and will take precedence over any in-progress interrupt sources.

NMIs can be used to implement critical device features such as forced software reset or other functionality where the action should be executed in preference to all other running code with a minimum amount of latency.

9.2.3. Input Filtering and Detection

To reduce the possibility of noise or other transient signals causing unwanted device wake-ups, interrupts, and/or events via an external interrupt channel. A hardware signal filter can be enabled on individual channels. This filter provides a Majority-of-Three voter filter on the incoming signal, so that the input state is considered to be the majority vote of three subsequent samples of the pin input buffer. The possible sampled input and resulting filtered output when the filter is enabled is shown in [Table 9-1](#).

Table 9-1. Sampled Input and Resulting Filtered Output

Input Sample 1	Input Sample 2	Input Sample 3	Filtered Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

9.2.4. Events and Interrupts

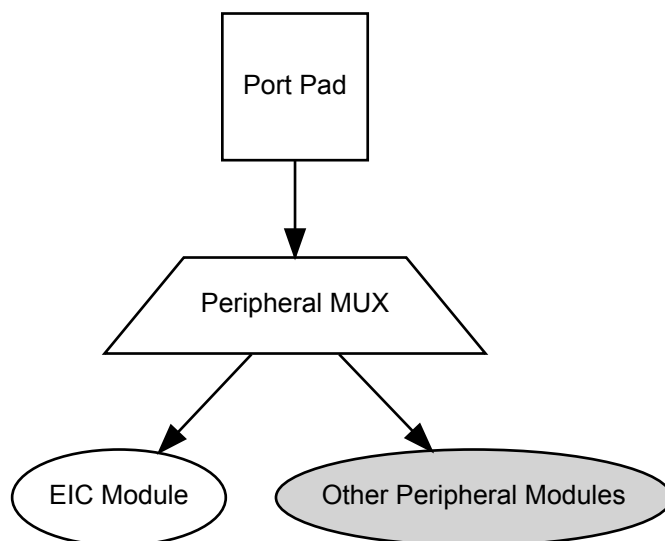
Channel detection states may be polled inside the application for synchronous detection, or events and interrupts may be used for asynchronous behavior. Each channel can be configured to give an asynchronous hardware event (which may in turn trigger actions in other hardware modules) or an asynchronous software interrupt.

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the input event of another. For more information on event routing, refer to the event driver documentation.

9.2.5. Physical Connection

[Figure 9-1](#) shows how this module is interconnected within the device.

Figure 9-1. Physical Connection



9.3. Special Considerations

Not all devices support disabling of the NMI channel(s) detection mode - see your device datasheet.

9.4. Extra Information

For extra information, see [Extra Information for EXTINT Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

9.5. Examples

For a list of examples related to this driver, see [Examples for EXTINT Driver](#).

9.6. API Overview

9.6.1. Variable and Type Definitions

9.6.1.1. Type `extint_callback_t`

```
typedef void(* extint_callback_t )(void)
```

Type definition for an EXTINT module callback function

9.6.2. Structure Definitions

9.6.2.1. Struct `extint_chan_conf`

Configuration structure for the edge detection mode of an external interrupt channel.

Table 9-2. Members

Type	Name	Description
enum extint_detect	<code>detection_criteria</code>	Edge detection mode to use
bool	<code>filter_input_signal</code>	Filter the raw input signal to prevent noise from triggering an interrupt accidentally, using a three sample majority filter
uint32_t	<code>gpio_pin</code>	GPIO pin the NMI should be connected to
uint32_t	<code>gpio_pin_mux</code>	MUX position the GPIO pin should be configured to
enum extint_pull	<code>gpio_pin_pull</code>	Internal pull to enable on the input pin
bool	<code>wake_if_sleeping</code>	Wake up the device if the channel interrupt fires during sleep mode

9.6.2.2. Struct `extint_events`

Event flags for the [extint_enable_events\(\)](#) and [extint_disable_events\(\)](#).

Table 9-3. Members

Type	Name	Description
bool	<code>generate_event_on_detect[]</code>	If <code>true</code> , an event will be generated when an external interrupt channel detection state changes

9.6.2.3. Struct `extint_nmi_conf`

Configuration structure for the edge detection mode of an external interrupt NMI channel.

Table 9-4. Members

Type	Name	Description
enum extint_detect	<code>detection_criteria</code>	Edge detection mode to use. Not all devices support all possible detection modes for NMIs.
bool	<code>filter_input_signal</code>	Filter the raw input signal to prevent noise from triggering an interrupt accidentally, using a three sample majority filter
uint32_t	<code>gpio_pin</code>	GPIO pin the NMI should be connected to
uint32_t	<code>gpio_pin_mux</code>	MUX position the GPIO pin should be configured to
enum extint_pull	<code>gpio_pin_pull</code>	Internal pull to enable on the input pin

9.6.3. Macro Definitions

9.6.3.1. Macro EIC_NUMBER_OF_INTERRUPTS

```
#define EIC_NUMBER_OF_INTERRUPTS
```

9.6.3.2. Macro EXTINT_CLK_GCLK

```
#define EXTINT_CLK_GCLK
```

The EIC is clocked by GCLK_EIC.

9.6.3.3. Macro EXTINT_CLK_ULP32K

```
#define EXTINT_CLK_ULP32K
```

The EIC is clocked by CLK_ULP32K.

9.6.4. Function Definitions

9.6.4.1. Event Management

Function `extint_enable_events()`

Enables an External Interrupt event output.

```
void extint_enable_events(  
    struct extint_events *const events)
```

Enables one or more output events from the External Interrupt module. See [here](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 9-5. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to enable

Function `extint_disable_events()`

Disables an External Interrupt event output.

```
void extint_disable_events(  
    struct extint_events *const events)
```

Disables one or more output events from the External Interrupt module. See [here](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 9-6. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to disable

9.6.4.2. Configuration and Initialization (Channel)

Function `extint_chan_get_config_defaults()`

Initializes an External Interrupt channel configuration structure to defaults.

```
void extint_chan_get_config_defaults(  
    struct extint_chan_conf *const config)
```

Initializes a given External Interrupt channel configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Wake the device if an edge detection occurs whilst in sleep
- Input filtering disabled
- Internal pull-up enabled
- Detect falling edges of a signal

Table 9-7. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `extint_chan_set_config()`

Writes an External Interrupt channel configuration to the hardware module.

```
void extint_chan_set_config(  
    const uint8_t channel,  
    const struct extint_chan_conf *const config)
```

Writes out a given configuration of an External Interrupt channel configuration to the hardware module. If the channel is already configured, the new configuration will replace the existing one.

Table 9-8. Parameters

Data direction	Parameter name	Description
[in]	channel	External Interrupt channel to configure
[in]	config	Configuration settings for the channel

9.6.4.3. Configuration and Initialization (NMI)

Function `extint_nmi_get_config_defaults()`

Initializes an External Interrupt NMI channel configuration structure to defaults.

```
void extint_nmi_get_config_defaults(  
    struct extint_nmi_conf *const config)
```

Initializes a given External Interrupt NMI channel configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Input filtering disabled

- Detect falling edges of a signal
- Asynchronous edge detection is disabled

Table 9-9. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `extint_nmi_set_config()`

Writes an External Interrupt NMI channel configuration to the hardware module.

```
enum status_code extint_nmi_set_config(
    const uint8_t nmi_channel,
    const struct extint_nmi_conf *const config)
```

Writes out a given configuration of an External Interrupt NMI channel configuration to the hardware module. If the channel is already configured, the new configuration will replace the existing one.

Table 9-10. Parameters

Data direction	Parameter name	Description
[in]	nmi_channel	External Interrupt NMI channel to configure
[in]	config	Configuration settings for the channel

Returns

Status code indicating the success or failure of the request.

Table 9-11. Return Values

Return value	Description
STATUS_OK	Configuration succeeded
STATUS_ERR_PIN_MUX_INVALID	An invalid pinmux value was supplied
STATUS_ERR_BAD_FORMAT	An invalid detection mode was requested

9.6.4.4. Detection testing and clearing (channel)

Function `extint_chan_is_detected()`

Retrieves the edge detection state of a configured channel.

```
bool extint_chan_is_detected(
    const uint8_t channel)
```

Reads the current state of a configured channel, and determines if the detection criteria of the channel has been met.

Table 9-12. Parameters

Data direction	Parameter name	Description
[in]	channel	External Interrupt channel index to check

Returns

Status of the requested channel's edge detection state.

Table 9-13. Return Values

Return value	Description
true	If the channel's edge/level detection criteria was met
false	If the channel has not detected its configured criteria

Function `extint_chan_clear_detected()`

Clears the edge detection state of a configured channel.

```
void extint_chan_clear_detected(  
    const uint8_t channel)
```

Clears the current state of a configured channel, readying it for the next level or edge detection.

Table 9-14. Parameters

Data direction	Parameter name	Description
[in]	channel	External Interrupt channel index to check

9.6.4.5. Detection Testing and Clearing (NMI)

Function `extint_nmi_is_detected()`

Retrieves the edge detection state of a configured NMI channel.

```
bool extint_nmi_is_detected(  
    const uint8_t nmi_channel)
```

Reads the current state of a configured NMI channel, and determines if the detection criteria of the NMI channel has been met.

Table 9-15. Parameters

Data direction	Parameter name	Description
[in]	nmi_channel	External Interrupt NMI channel index to check

Returns

Status of the requested NMI channel's edge detection state.

Table 9-16. Return Values

Return value	Description
true	If the NMI channel's edge/level detection criteria was met
false	If the NMI channel has not detected its configured criteria

Function `extint_nmi_clear_detected()`

Clears the edge detection state of a configured NMI channel.

```
void extint_nmi_clear_detected(  
    const uint8_t nmi_channel)
```

Clears the current state of a configured NMI channel, readying it for the next level or edge detection.

Table 9-17. Parameters

Data direction	Parameter name	Description
[in]	nmi_channel	External Interrupt NMI channel index to check

9.6.4.6. Callback Configuration and Initialization

Function `extint_register_callback()`

Registers an asynchronous callback function with the driver.

```
enum status_code extint_register_callback(  
    const extint_callback_t callback,  
    const uint8_t channel,  
    const enum extint_callback_type type)
```

Registers an asynchronous callback with the EXTINT driver, fired when a channel detects the configured channel detection criteria (e.g. edge or level). Callbacks are fired once for each detected channel.

Note: NMI channel callbacks cannot be registered via this function; the device's NMI interrupt should be hooked directly in the user application and the NMI flags manually cleared via [extint_nmi_clear_detected\(\)](#).

Table 9-18. Parameters

Data direction	Parameter name	Description
[in]	callback	Pointer to the callback function to register
[in]	channel	Logical channel to register callback for
[in]	type	Type of callback function to register

Returns

Status of the registration operation.

Table 9-19. Return Values

Return value	Description
STATUS_OK	The callback was registered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied
STATUS_ERR_ALREADY_INITIALIZED	Callback function has been registered, need unregister first

Function `extint_unregister_callback()`

Unregisters an asynchronous callback function with the driver.

```
enum status_code extint_unregister_callback(  
    const extint_callback_t callback,  
    const uint8_t channel,  
    const enum extint_callback_type type)
```

Unregisters an asynchronous callback with the EXTINT driver, removing it from the internal callback registration table.

Table 9-20. Parameters

Data direction	Parameter name	Description
[in]	callback	Pointer to the callback function to unregister
[in]	channel	Logical channel to unregister callback for
[in]	type	Type of callback function to unregister

Returns

Status of the de-registration operation.

Table 9-21. Return Values

Return value	Description
STATUS_OK	The callback was unregistered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied
STATUS_ERR_BAD_ADDRESS	No matching entry was found in the registration table

Function `extint_get_current_channel()`

Find what channel caused the callback.

```
uint8_t extint_get_current_channel( void )
```

Can be used in an EXTINT callback function to find what channel caused the callback in case the same callback is used by multiple channels.

Returns

Channel number.

9.6.4.7. Callback Enabling and Disabling (Channel)

Function `extint_chan_enable_callback()`

Enables asynchronous callback generation for a given channel and type.

```
enum status_code extint_chan_enable_callback(  
    const uint8_t channel,  
    const enum extint_callback_type type)
```

Enables asynchronous callbacks for a given logical external interrupt channel and type. This must be called before an external interrupt channel will generate callback events.

Table 9-22. Parameters

Data direction	Parameter name	Description
[in]	channel	Logical channel to enable callback generation for
[in]	type	Type of callback function callbacks to enable

Returns

Status of the callback enable operation.

Table 9-23. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function extint_chan_disable_callback()

Disables asynchronous callback generation for a given channel and type.

```
enum status_code extint_chan_disable_callback(
    const uint8_t channel,
    const enum extint_callback_type type)
```

Disables asynchronous callbacks for a given logical external interrupt channel and type.

Table 9-24. Parameters

Data direction	Parameter name	Description
[in]	channel	Logical channel to disable callback generation for
[in]	type	Type of callback function callbacks to disable

Returns

Status of the callback disable operation.

Table 9-25. Return Values

Return value	Description
STATUS_OK	The callback was disabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

9.6.5. Enumeration Definitions**9.6.5.1. Callback Configuration and Initialization****Enum extint_callback_type**

Enum for the possible callback types for the EXTINT module.

Table 9-26. Members

Enum value	Description
EXTINT_CALLBACK_TYPE_DETECT	Callback type for when an external interrupt detects the configured channel criteria (i.e. edge or level detection)

9.6.5.2. Enum `extint_detect`

Enum for the possible signal edge detection modes of the External Interrupt Controller module.

Table 9-27. Members

Enum value	Description
EXTINT_DETECT_NONE	No edge detection. Not allowed as a NMI detection mode on some devices.
EXTINT_DETECT_RISING	Detect rising signal edges
EXTINT_DETECT_FALLING	Detect falling signal edges
EXTINT_DETECT_BOTH	Detect both signal edges
EXTINT_DETECT_HIGH	Detect high signal levels
EXTINT_DETECT_LOW	Detect low signal levels

9.6.5.3. Enum `extint_pull`

Enum for the possible pin internal pull configurations.

Note: Disabling the internal pull resistor is not recommended if the driver is used in interrupt (callback) mode, due the possibility of floating inputs generating continuous interrupts.

Table 9-28. Members

Enum value	Description
EXTINT_PULL_UP	Internal pull-up resistor is enabled on the pin
EXTINT_PULL_DOWN	Internal pull-down resistor is enabled on the pin
EXTINT_PULL_NONE	Internal pull resistor is disconnected from the pin

9.7. Extra Information for EXTINT Driver

9.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
EIC	External Interrupt Controller
MUX	Multiplexer
NMI	Non-Maskable Interrupt

9.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

9.7.3. Errata

There are no errata related to this driver.

9.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
• Driver updated to follow driver type convention • Removed <code>extint_reset()</code>, <code>extint_disable()</code> and <code>extint_enable()</code> functions. Added internal function <code>_system_extint_init()</code>. • Added configuration <code>EXTINT_CLOCK_SOURCE</code> in <code>conf_extint.h</code> • Removed configuration <code>EXTINT_CALLBACKS_MAX</code> in <code>conf_extint.h</code>, and added channel parameter in the register functions <code>extint_register_callback()</code> and <code>extint_unregister_callback()</code>
Updated interrupt handler to clear interrupt flag before calling callback function
Updated initialization function to also enable the digital interface clock to the module if it is disabled
Initial Release

9.8. Examples for EXTINT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM External Interrupt \(EXTINT\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for EXTINT - Basic](#)
- [Quick Start Guide for EXTINT - Callback](#)

9.8.1. Quick Start Guide for EXTINT - Basic

The supported board list:

- SAM D20 Xplained Pro
- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM DA1 Xplained Pro
- SAM C21 Xplained Pro

In this use case, the EXTINT module is configured for:

- External interrupt channel connected to the board LED is used
- External interrupt channel is configured to detect both input signal edges

This use case configures a physical I/O pin of the device so that it is routed to a logical External Interrupt Controller channel to detect rising and falling edges of the incoming signal.

When the board button is pressed, the board LED will light up. When the board button is released, the LED will turn off.

9.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_extint_channel(void)
{
    struct extint_chan_conf config_extint_chan;
    extint_chan_get_config_defaults(&config_extint_chan);

    config_extint_chan.gpio_pin          = BUTTON_0_EIC_PIN;
    config_extint_chan.gpio_pin_mux      = BUTTON_0_EIC_MUX;
    config_extint_chan.gpio_pin_pull     = EXTINT_PULL_UP;
    config_extint_chan.detection_criteria = EXTINT_DETECT_BOTH;
    extint_chan_set_config(BUTTON_0_EIC_LINE, &config_extint_chan);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_extint_channel();
```

Workflow

1. Create an EXTINT module channel configuration struct, which can be filled out to adjust the configuration of a single external interrupt channel.

```
struct extint_chan_conf config_extint_chan;
```

2. Initialize the channel configuration struct with the module's default values.

```
extint_chan_get_config_defaults(&config_extint_chan);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to configure the pin MUX (to route the desired physical pin to the logical channel) to the board button, and to configure the channel to detect both rising and falling edges.

```
config_extint_chan.gpio_pin          = BUTTON_0_EIC_PIN;  
config_extint_chan.gpio_pin_mux     = BUTTON_0_EIC_MUX;  
config_extint_chan.gpio_pin_pull    = EXTINT_PULL_UP;  
config_extint_chan.detection_criteria = EXTINT_DETECT_BOTH;
```

4. Configure external interrupt channel with the desired channel settings.

```
extint_chan_set_config(BUTTON_0_EIC_LINE, &config_extint_chan);
```

9.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    if (extint_chan_is_detected(BUTTON_0_EIC_LINE)) {  
  
        // Do something in response to EXTINT edge detection  
        bool button_pin_state = port_pin_get_input_level(BUTTON_0_PIN);  
        port_pin_set_output_level(LED_0_PIN, button_pin_state);  
  
        extint_chan_clear_detected(BUTTON_0_EIC_LINE);  
    }  
}
```

Workflow

1. Read in the current external interrupt channel state to see if an edge has been detected.

```
if (extint_chan_is_detected(BUTTON_0_EIC_LINE)) {
```

2. Read in the new physical button state and mirror it on the board LED.

```
    // Do something in response to EXTINT edge detection  
    bool button_pin_state = port_pin_get_input_level(BUTTON_0_PIN);  
    port_pin_set_output_level(LED_0_PIN, button_pin_state);
```

3. Clear the detection state of the external interrupt channel so that it is ready to detect a future falling edge.

```
    extint_chan_clear_detected(BUTTON_0_EIC_LINE);
```

9.8.2. Quick Start Guide for EXTINT - Callback

The supported board list:

- SAM D20 Xplained Pro
- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM DA1 Xplained Pro
- SAM C21 Xplained Pro

In this use case, the EXTINT module is configured for:

- External interrupt channel connected to the board LED is used

- External interrupt channel is configured to detect both input signal edges
- Callbacks are used to handle detections from the External Interrupt

This use case configures a physical I/O pin of the device so that it is routed to a logical External Interrupt Controller channel to detect rising and falling edges of the incoming signal. A callback function is used to handle detection events from the External Interrupt module asynchronously.

When the board button is pressed, the board LED will light up. When the board button is released, the LED will turn off.

9.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_extint_channel(void)
{
    struct extint_chan_conf config_extint_chan;
    extint_chan_get_config_defaults(&config_extint_chan);

    config_extint_chan.gpio_pin          = BUTTON_0_EIC_PIN;
    config_extint_chan.gpio_pin_mux      = BUTTON_0_EIC_MUX;
    config_extint_chan.gpio_pin_pull     = EXTINT_PULL_UP;
    config_extint_chan.detection_criteria = EXTINT_DETECT_BOTH;
    extint_chan_set_config(BUTTON_0_EIC_LINE, &config_extint_chan);
}

void configure_extint_callbacks(void)
{
    extint_register_callback(extint_detection_callback,
        BUTTON_0_EIC_LINE,
        EXTINT_CALLBACK_TYPE_DETECT);
    extint_chan_enable_callback(BUTTON_0_EIC_LINE,
        EXTINT_CALLBACK_TYPE_DETECT);
}

void extint_detection_callback(void)
{
    bool pin_state = port_pin_get_input_level(BUTTON_0_PIN);
    port_pin_set_output_level(LED_0_PIN, pin_state);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_extint_channel();
configure_extint_callbacks();

system_interrupt_enable_global();
```

Workflow

1. Create an EXTINT module channel configuration struct, which can be filled out to adjust the configuration of a single external interrupt channel.

```
struct extint_chan_conf config_extint_chan;
```

2. Initialize the channel configuration struct with the module's default values.

```
extint_chan_get_config_defaults(&config_extint_chan);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to configure the pin MUX (to route the desired physical pin to the logical channel) to the board button, and to configure the channel to detect both rising and falling edges.

```
config_extint_chan.gpio_pin      = BUTTON_0_EIC_PIN;
config_extint_chan.gpio_pin_mux  = BUTTON_0_EIC_MUX;
config_extint_chan.gpio_pin_pull = EXTINT_PULL_UP;
config_extint_chan.detection_criteria = EXTINT_DETECT_BOTH;
```

4. Configure external interrupt channel with the desired channel settings.

```
extint_chan_set_config(BUTTON_0_EIC_LINE, &config_extint_chan);
```

5. Register a callback function `extint_handler()` to handle detections from the External Interrupt controller.

```
extint_register_callback(extint_detection_callback,
    BUTTON_0_EIC_LINE,
    EXTINT_CALLBACK_TYPE_DETECT);
```

6. Enable the registered callback function for the configured External Interrupt channel, so that it will be called by the module when the channel detects an edge.

```
extint_chan_enable_callback(BUTTON_0_EIC_LINE,
    EXTINT_CALLBACK_TYPE_DETECT);
```

7. Define the EXTINT callback that will be fired when a detection event occurs. For this example, a LED will mirror the new button state on each detection edge.

```
void extint_detection_callback(void)
{
    bool pin_state = port_pin_get_input_level(BUTTON_0_PIN);
    port_pin_set_output_level(LED_0_PIN, pin_state);
}
```

9.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {
    /* Do nothing - EXTINT will fire callback asynchronously */
}
```

Workflow

1. External interrupt events from the driver are detected asynchronously; no special application `main()` code is required.

10. SAM Frequency Meter (FREQM) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Frequency Meter functionality.

The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- FREQM (Frequency Meter)

The following devices can use this module:

- Atmel | SMART SAM L22
- Atmel | SMART SAM C20
- Atmel | SMART SAM C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

10.1. Prerequisites

There are no prerequisites for this module.

10.2. Module Overview

The module accurately measures the frequency of a clock by comparing it to a known reference clock as soon as the FREQM is enabled. Two generic clocks are used by the FREQM. The frequency of the measured clock is:

$$f_{CLK\_MSR} = \frac{VALUE}{REFNUM} \times f_{CLK\_REF}$$

Ratio can be measured with 24-bit accuracy.

The FREQM has one interrupt source, which generates when a frequency measurement is done. It can be used to wake up the device from sleep modes.

This driver provides an interface for the FREQM functions on the device.

10.3. Special Considerations

There are no special considerations for this module.

10.4. Extra Information

For extra information see [Extra Information for FREQM Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

10.5. Examples

For a list of examples related to this driver, see [Examples for FREQM Driver](#).

10.6. API Overview

10.6.1. Variable and Type Definitions

10.6.1.1. Type `freqm_callback_t`

```
typedef void(* freqm_callback_t )(void)
```

Type definition for a FREQM module callback function.

10.6.1.2. Variable `_freqm_instance`

```
struct freqm_module * _freqm_instance
```

10.6.2. Structure Definitions

10.6.2.1. Struct `freqm_config`

Configuration structure for a Frequency Meter.

Table 10-1. Members

Type	Name	Description
enum gclk_generator	msr_clock_source	GCLK source select for measurement
uint16_t	ref_clock_circles	Measurement duration in number of reference clock cycles. Range 1~255
enum gclk_generator	ref_clock_source	GCLK source select for reference

10.6.2.2. Struct `freqm_module`

FREQM software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

10.6.3. Function Definitions

10.6.3.1. Driver Initialization and Configuration

Function `freqm_init()`

Initializes a hardware FREQM module instance.

```
enum status_code freqm_init(  
    struct freqm_module *const module_inst,  
    Freqm *const hw,  
    struct freqm_config *const config)
```

Enables the clock and initializes the FREQM module, based on the given configuration values.

Table 10-2. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	hw	Pointer to the FREQM hardware module
[in]	config	Pointer to the FREQM configuration options struct

Returns

Status of the initialization procedure.

Table 10-3. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully

Function `freqm_get_config_defaults()`

Initializes all members of a FREQM configuration structure to safe defaults.

```
void freqm_get_config_defaults(  
    struct freqm_config *const config)
```

Initializes all members of a given Frequency Meter configuration structure to safe known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Measurement clock source is GCLK0
- Reference clock source is GCLK1
- Frequency Meter Reference Clock Cycles 127

Table 10-4. Parameters

Data direction	Parameter name	Description
[in]	config	Configuration structure to initialize to default values

Function `freqm_enable()`

Enables a FREQM that was previously configured.

```
void freqm_enable(  
    struct freqm_module *const module_inst)
```

Enables Frequency Meter that was previously configured via a call to `freqm_init()`.

Table 10-5. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Frequency Meter peripheral

Function `freqm_disable()`

Disables a FREQM that was previously enabled.

```
void freqm_disable(  
    struct freqm_module *const module_inst)
```

Disables Frequency Meter that was previously started via a call to `freqm_enable()`.

Table 10-6. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the Frequency Meter peripheral

10.6.3.2. Read FREQM Result

Function `freqm_start_measure()`

Start a manual measurement process.

```
void freqm_start_measure(  
    struct freqm_module *const module)
```

Table 10-7. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the FREQM software instance struct

Function `freqm_clear_overflow()`

Clears module overflow flag.

```
void freqm_clear_overflow(  
    struct freqm_module *const module)
```

Clears the overflow flag of the module.

Table 10-8. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the FREQM software instance struct

Function `freqm_get_result_value()`

Read the measurement data result.

```
enum freqm_status freqm_get_result_value(  
    struct freqm_module *const module_inst,  
    uint32_t * result)
```

Reads the measurement data result.

Table 10-9. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the FREQM software instance struct
[out]	result	Pointer to store the result value in

Note: If overflow occurred, configure faster reference clock or reduce reference clock cycles.

Returns

Status of the FREQM read request.

Table 10-10. Return Values

Return value	Description
FREQM_STATUS_MEASURE_DONE	Measurement result was retrieved successfully
FREQM_STATUS_MEASURE_BUSY	Measurement result was not ready
FREQM_STATUS_CNT_OVERFLOW	Measurement result was overflow

10.6.3.3. Callback Configuration and Initialization

Function `freqm_register_callback()`

Registers a callback.

```
enum status_code freqm_register_callback(  
    struct freqm_module *const module,  
    freqm_callback_t callback_func,  
    enum freqm_callback callback_type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled by `freqm_enable_callback`, in order for the interrupt handler to call it when the conditions for the callback type is met.

Table 10-11. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to FREQM software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

Table 10-12. Return Values

Return value	Description
STATUS_OK	The function exited successfully

Function freqm_unregister_callback()

Unregisters a callback.

```
enum status_code freqm_unregister_callback(
    struct freqm_module * module,
    enum freqm_callback callback_type)
```

Unregisters a callback function implemented by the user. The callback should be disabled before it is unregistered.

Table 10-13. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to FREQM software instance struct
[in]	callback_type	Callback type given by an enum

Table 10-14. Return Values

Return value	Description
STATUS_OK	The function exited successfully

10.6.3.4. Callback Enabling and Disabling**Function freqm_enable_callback()**

Enable an FREQM callback.

```
enum status_code freqm_enable_callback(
    struct freqm_module *const module,
    const enum freqm_callback_type type)
```

Table 10-15. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	type	Callback source type

Returns

Status of the callback enable operation.

Table 10-16. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function `freqm_disable_callback()`

Disable an FREQM callback.

```
enum status_code freqm_disable_callback(  
    struct freqm_module *const module,  
    const enum freqm_callback_type type)
```

Table 10-17. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	type	Callback source type

Returns

Status of the callback enable operation.

Table 10-18. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

10.6.3.5. Function `freqm_is_syncing()`

Determines if the hardware module(s) are currently synchronizing to the bus.

```
bool freqm_is_syncing( void )
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Returns

Synchronization status of the underlying hardware module(s).

Table 10-19. Return Values

Return value	Description
false	If the module has completed synchronization
true	If the module synchronization is ongoing

10.6.4. Enumeration Definitions

10.6.4.1. Enum `freqm_callback`

Enum for possible callback types for the FREQM module.

Table 10-20. Members

Enum value	Description
FREQM_CALLBACK_DONE	Callback for measurement done

10.6.4.2. Enum freqm_callback_type

FREQM callback type.

Table 10-21. Members

Enum value	Description
FREQM_CALLBACK_MEASURE_DONE	Measurement done callback.

10.6.4.3. Enum freqm_status

Enum for the possible status types for the FREQM module.

Table 10-22. Members

Enum value	Description
FREQM_STATUS_MEASURE_DONE	FREQM measurement is finish
FREQM_STATUS_MEASURE_BUSY	FREQM measurement is ongoing or not
FREQM_STATUS_CNT_OVERFLOW	FREQM sticky count value overflow

10.7. Extra Information for FREQM Driver

10.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
FREQM	Frequency Meter

10.7.2. Dependencies

This driver has no dependencies.

10.7.3. Errata

There are no errata related to this driver.

10.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

10.8. Examples for FREQM Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Frequency Meter \(FREQM\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use

this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for FREQM - Basic](#)
- [Quick Start Guide for FREQM - Callback](#)

10.8.1. Quick Start Guide for FREQM - Basic

In this use case, the Frequency Meter (FREQM) module is configured for:

- The FREQM peripheral will not be stopped in standby sleep mode.

This use case will read measurement data in polling mode repeatedly. After reading a data, the board LED will be toggled.

10.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
/* FREQM module software instance (must not go out of scope while in use) */
static struct freqm_module freqm_instance;

void configure_freqm(void)
{
    /* Create a new configuration structure for the FREQM settings
    * and fill with the default module settings. */
    struct freqm_config config_freqm;
    freqm_get_config_defaults(&config_freqm);

    /* Alter any FREQM configuration settings here if required */

    /* Initialize FREQM with the user settings */
    freqm_init(&freqm_instance, FREQM, &config_freqm);
}
```

Add to user application initialization (typically the start of `main()`):

```
system_init();
configure_freqm();
freqm_enable(&freqm_instance);
```

Workflow

1. Create an FREQM device instance struct, which will be associated with a FREQM peripheral hardware instance.

```
static struct freqm_module freqm_instance;
```

Note: Device instance structures shall **never** go out of scope when in use.

2. Create a new function `configure_freqm()`, which will be used to configure the overall FREQM peripheral.

```
void configure_freqm(void)
```

3. Create an FREQM peripheral configuration structure that will be filled out to set the module configuration.

```
struct freqm_config config_freqm;
```

4. Fill the FREQM peripheral configuration structure with the default module configuration values.

```
freqm_get_config_defaults(&config_freqm);
```

5. Initialize the FREQM peripheral and associate it with the software instance structure that was defined previously.

```
freqm_init(&freqm_instance, FREQM, &config_freqm);
```

6. Enable the now initialized FREQM peripheral.

```
freqm_enable(&freqm_instance);
```

10.8.1.2. Implementation

Code

Copy-paste the following code to your user application:

```
uint32_t measure_result;
enum freqm_status status;

freqm_start_measure(&freqm_instance);
while ((status = freqm_get_result_value(&freqm_instance, &measure_result))
      == FREQM_STATUS_MEASURE_BUSY) {
};
switch(status) {
    case FREQM_STATUS_MEASURE_DONE:
        LED_On(LED_0_PIN);
        while (true) {
        }
    case FREQM_STATUS_CNT_OVERFLOW:
        freqm_clear_overflow(&freqm_instance);
        while (true) {
            LED_Toggle(LED_0_PIN);
            volatile uint32_t delay = 50000;
            while(delay--) {
            }
        }
    default:
        Assert(false);
        break;
}
```

Workflow

1. Start FREQM measurement and wait until measure done then read result data.

```
freqm_start_measure(&freqm_instance);
while ((status = freqm_get_result_value(&freqm_instance,
&measure_result))
      == FREQM_STATUS_MEASURE_BUSY) {
};
```

2. The board LED is on to indicate a measurement data is read.

```
case FREQM_STATUS_MEASURE_DONE:
    LED_On(LED_0_PIN);
    while (true) {
    }
```


3. The board LED is toggle to indicate measurement is overflow.

```
case FREQM_STATUS_CNT_OVERFLOW:
    freqm_clear_overflow(&freqm_instance);
    while (true) {
        LED_Toggle(LED_0_PIN);
        volatile uint32_t delay = 50000;
        while(delay--) {
        }
    }
}
```

10.8.2. Quick Start Guide for FREQM - Callback

In this use case, the Frequency Meter (FREQM) module is configured for:

- The FREQM peripheral will not be stopped in standby sleep mode.

This use case will read measurement data in interrupt mode repeatedly. After reading specific size of buffer data, the board LED will be toggled.

10.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
bool volatile freqm_read_done = false;

void configure_freqm(void);
void configure_freqm_callback(void);
void freqm_complete_callback(void);

/* FREQM module software instance (must not go out of scope while in use) */
static struct freqm_module freqm_instance;

void configure_freqm(void)
{
    /* Create a new configuration structure for the FREQM settings
    * and fill with the default module settings. */
    struct freqm_config config_freqm;
    freqm_get_config_defaults(&config_freqm);
    config_freqm.ref_clock_circles = 255;

    /* Alter any FREQM configuration settings here if required */

    /* Initialize FREQM with the user settings */
    freqm_init(&freqm_instance, FREQM, &config_freqm);
}

void freqm_complete_callback(void)
{
    freqm_read_done = true;
}

void configure_freqm_callback(void)
{
    freqm_register_callback(&freqm_instance, freqm_complete_callback,
        FREQM_CALLBACK_MEASURE_DONE);
    freqm_enable_callback(&freqm_instance, FREQM_CALLBACK_MEASURE_DONE);
}
```

Add to user application initialization (typically the start of `main()`):

```
system_init();
configure_freqm();
configure_freqm_callback();

freqm_enable(&freqm_instance);
```

Workflow

1. Create an FREQM device instance struct, which will be associated with an FREQM peripheral hardware instance.

```
static struct freqm_module freqm_instance;
```

Note: Device instance structures shall **never** go out of scope when in use.

2. Create a new function `configure_freqm()`, which will be used to configure the overall FREQM peripheral.

```
void configure_freqm(void)
```

3. Create an FREQM peripheral configuration structure that will be filled out to set the module configuration.

```
struct freqm_config config_freqm;
```

4. Fill the FREQM peripheral configuration structure with the default module configuration values.

```
freqm_get_config_defaults(&config_freqm);
config_freqm.ref_clock_circles = 255;
```

5. Initialize the FREQM peripheral and associate it with the software instance structure that was defined previously.

```
freqm_init(&freqm_instance, FREQM, &config_freqm);
```

6. Create a new callback function.

```
void freqm_complete_callback(void)
{
    freqm_read_done = true;
}
```

7. Create a callback status software flag.

```
bool volatile freqm_read_done = false;
```

8. Let the callback function set the flag to true when read job done.

```
freqm_read_done = true;
```

9. Create a new function `configure_freqm_callback()`, which will be used to configure the callbacks.

```
void configure_freqm_callback(void)
{
    freqm_register_callback(&freqm_instance, freqm_complete_callback,
        FREQM_CALLBACK_MEASURE_DONE);
    freqm_enable_callback(&freqm_instance,
        FREQM_CALLBACK_MEASURE_DONE);
}
```

10. Register callback function.

```
freqm_register_callback(&freqm_instance, freqm_complete_callback,  
    FREQM_CALLBACK_MEASURE_DONE);
```

11. Enable the callbacks.

```
freqm_enable_callback(&freqm_instance, FREQM_CALLBACK_MEASURE_DONE);
```

12. Enable the now initialized FREQM peripheral.

```
freqm_enable(&freqm_instance);
```

Note: This should not be done until after the FREQM is setup and ready to be used.

10.8.2.2. Implementation

Code

Copy-paste the following code to your user application:

```
uint32_t measure_result;  
enum freqm_status status;  
freqm_start_measure(&freqm_instance);  
  
while (!freqm_read_done) {  
}  
status = freqm_get_result_value(&freqm_instance, &measure_result);  
switch(status) {  
    case FREQM_STATUS_MEASURE_DONE:  
        LED_On(LED_0_PIN);  
        while (true) {  
        }  
    case FREQM_STATUS_CNT_OVERFLOW:  
        freqm_clear_overflow(&freqm_instance);  
        while (true) {  
            LED_Toggle(LED_0_PIN);  
            volatile uint32_t delay = 50000;  
            while(delay--) {  
            }  
        }  
    default:  
        Assert(false);  
        break;  
}
```

Workflow

1. Start an asynchronous FREQM read job, to store measurement data into the global buffer and generate a callback when complete.

```
freqm_start_measure(&freqm_instance);
```

2. Wait until the asynchronous read job is complete.

```
while (!freqm_read_done) {  
}  
status = freqm_get_result_value(&freqm_instance, &measure_result);
```

3. The board LED on to indicate measurement data read.

```
case FREQM_STATUS_MEASURE_DONE:  
    LED_On(LED_0_PIN);  
    while (true) {  
    }  
}
```

4. The board LED toggled to indicate measurement overflow occurs.

```
case FREQM_STATUS_CNT_OVERFLOW:
    freqm_clear_overflow(&freqm_instance);
    while (true) {
        LED_Toggle(LED_0_PIN);
        volatile uint32_t delay = 50000;
        while(delay--) {
        }
    }
}
```

11. SAM Non-Volatile Memory (NVM) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of non-volatile memories within the device, for partitioning, erasing, reading, and writing of data.

The following peripheral is used by this module:

- NVM (Non-Volatile Memory)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

11.1. Prerequisites

There are no prerequisites for this module.

11.2. Module Overview

The Non-Volatile Memory (NVM) module provides an interface to the device's Non-Volatile Memory controller, so that memory pages can be written, read, erased, and reconfigured in a standardized manner.

11.2.1. Driver Feature Macro Definition

Driver feature macro	Supported devices
FEATURE_NVM_RWWEE	SAM L21/L22, SAM D21-64K, SAM DA1, SAM C20/C21, SAM R30
FEATURE_BOD12	SAM L21, SAMR30

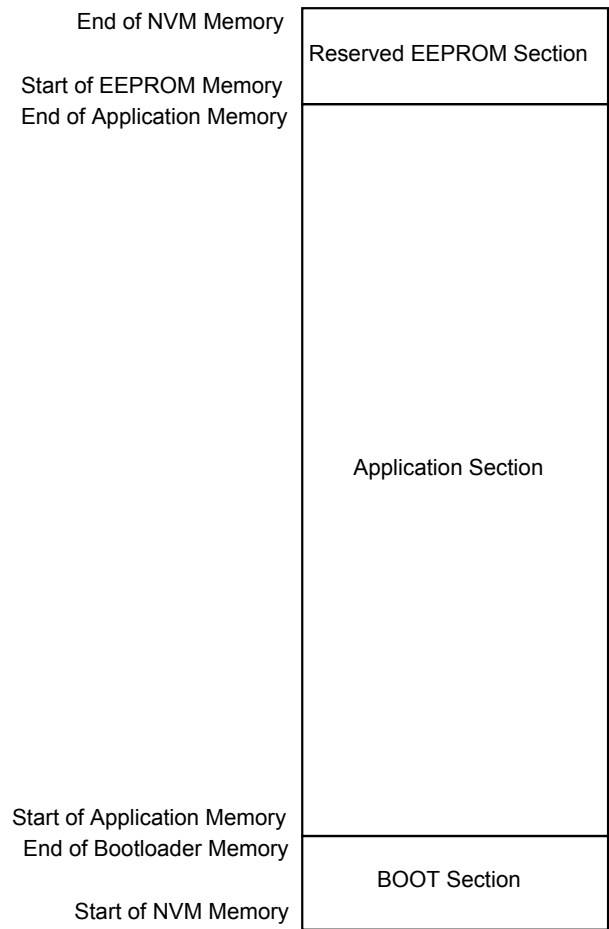
Note: The specific features are only available in the driver when the selected device supports those features.

11.2.2. Memory Regions

The NVM memory space of the SAM devices is divided into two sections: a Main Array section, and an Auxiliary space section. The Main Array space can be configured to have an (emulated) EEPROM and/or

boot loader section. The memory layout with the EEPROM and bootloader partitions is shown in [Figure 11-1](#).

Figure 11-1. Memory Regions



The Main Array is divided into rows and pages, where each row contains four pages. The size of each page may vary from 8-1024 bytes dependent of the device. Device specific parameters such as the page size and total number of pages in the NVM memory space are available via the [nvm_get_parameters\(\)](#) function.

An NVM page number and address can be computed via the following equations:

$$\begin{aligned} \text{PageNum} &= (\text{RowNum} \times 4) + \text{PagePosInRow} \\ \text{PageAddr} &= \text{PageNum} \times \text{PageSize} \end{aligned}$$

[Figure 11-2](#) shows an example of the memory page and address values associated with logical row 7 of the NVM memory space.

Figure 11-2. Memory Regions

Row 0x07	Page 0x1F	Page 0x1E	Page 0x1D	Page 0x1C
Address	0x7C0	0x780	0x740	0x700

11.2.3. Region Lock Bits

As mentioned in [Memory Regions](#), the main block of the NVM memory is divided into a number of individually addressable pages. These pages are grouped into 16 equal sized regions, where each region can be locked separately issuing an [NVM_COMMAND_LOCK_REGION](#) command or by writing the LOCK bits in the User Row. Rows reserved for the EEPROM section are not affected by the lock bits or commands.

Note: By using the [NVM_COMMAND_LOCK_REGION](#) or [NVM_COMMAND_UNLOCK_REGION](#) commands the settings will remain in effect until the next device reset. By changing the default lock setting for the regions, the auxiliary space must be written, however the adjusted configuration will not take effect until the next device reset.

Note: If the [Security Bit](#) is set, the auxiliary space cannot be written to. Clearing of the security bit can only be performed by a full chip erase.

11.2.4. Read/Write

Reading from the NVM memory can be performed using direct addressing into the NVM memory space, or by calling the [nvm_read_buffer\(\)](#) function.

Writing to the NVM memory must be performed by the [nvm_write_buffer\(\)](#) function - additionally, a manual page program command must be issued if the NVM controller is configured in manual page writing mode.

Before a page can be updated, the associated NVM memory row must be erased first via the [nvm_erase_row\(\)](#) function. Writing to a non-erased page will result in corrupt data being stored in the NVM memory space.

11.3. Special Considerations

11.3.1. Page Erasure

The granularity of an erase is per row, while the granularity of a write is per page. Thus, if the user application is modifying only one page of a row, the remaining pages in the row must be buffered and the row erased, as an erase is mandatory before writing to a page.

11.3.2. Clocks

The user must ensure that the driver is configured with a proper number of wait states when the CPU is running at high frequencies.

11.3.3. Security Bit

The User Row in the Auxiliary Space cannot be read or written when the Security Bit is set. The Security Bit can be set by using passing [NVM_COMMAND_SET_SECURITY_BIT](#) to the [nvm_execute_command\(\)](#) function, or it will be set if one tries to access a locked region. See [Region Lock Bits](#).

The Security Bit can only be cleared by performing a chip erase.

11.4. Extra Information

For extra information, see [Extra Information for NVM Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

11.5. Examples

For a list of examples related to this driver, see [Examples for NVM Driver](#).

11.6. API Overview

11.6.1. Structure Definitions

11.6.1.1. Struct `nvm_config`

Configuration structure for the NVM controller within the device.

Table 11-1. Members

Type	Name	Description
enum nvm_cache_readmode	cache_readmode	Select the mode for how the cache will pre-fetch data from the flash
bool	disable_cache	Setting this to true will disable the pre-fetch cache in front of the NVM controller
bool	manual_page_write	Manual write mode; if enabled, pages loaded into the NVM buffer will not be written until a separate write command is issued. If disabled, writing to the last byte in the NVM page buffer will trigger an automatic write. Note: If a partial page is to be written, a manual write command must be executed in either mode.
enum nvm_sleep_power_mode	sleep_power_mode	Power reduction mode during device sleep
uint8_t	wait_states	Number of wait states to insert when reading from flash, to prevent invalid data from being read at high clock frequencies

11.6.1.2. Struct `nvm_fusebits`

This structure contain the layout of the first 64 bits of the user row which contain the fuse settings.

Table 11-2. Members

Type	Name	Description
enum nvm_bod12_action	bod12_action	BOD12 Action at power on
bool	bod12_enable	BOD12 Enable at power on
bool	bod12_hysteresis	
uint8_t	bod12_level	BOD12 Threshold level at power on
enum nvm_bod33_action	bod33_action	BOD33 Action at power on
bool	bod33_enable	BOD33 Enable at power on
bool	bod33_hysteresis	
uint8_t	bod33_level	BOD33 Threshold level at power on
enum nvm_bootloader_size	bootloader_size	Bootloader size
enum nvm_eeprom_emulator_size	eeprom_size	EEPROM emulation area size
uint16_t	lockbits	NVM Lock bits
bool	wdt_always_on	WDT Always-on at power on
enum nvm_wdt_early_warning_offset	wdt_early_warning_offset	WDT Early warning interrupt time offset at power on
bool	wdt_enable	WDT Enable at power on
uint8_t	wdt_timeout_period	WDT Period at power on
bool	wdt_window_mode_enable_at_poweron	WDT Window mode enabled at power on
enum nvm_wdt_window_timeout	wdt_window_timeout	WDT Window mode time-out at power on

11.6.1.3. Struct `nvm_parameters`

Structure containing the memory layout parameters of the NVM module.

Table 11-3. Members

Type	Name	Description
uint32_t	bootloader_number_of_pages	Size of the Bootloader memory section configured in the NVM auxiliary memory space
uint32_t	eeeprom_number_of_pages	Size of the emulated EEPROM memory section configured in the NVM auxiliary memory space
uint16_t	nvm_number_of_pages	Number of pages in the main array
uint8_t	page_size	Number of bytes per page
uint16_t	rww_eeeprom_number_of_pages	Number of pages in read while write EEPROM (RWWEE) emulation area

11.6.2. Macro Definitions

11.6.2.1. Driver Feature Definition

Define NVM features set according to the different device families.

Macro FEATURE_NVM_RWWEE

```
#define FEATURE_NVM_RWWEE
```

Read while write EEPROM emulation feature.

Macro FEATURE_BOD12

```
#define FEATURE_BOD12
```

Brown-out detector internal to the voltage regulator for VDDCORE.

11.6.3. Function Definitions

11.6.3.1. Configuration and Initialization

Function nvm_get_config_defaults()

Initializes an NVM controller configuration structure to defaults.

```
void nvm_get_config_defaults(  
    struct nvm_config *const config)
```

Initializes a given NVM controller configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Power reduction mode enabled after sleep mode until first NVM access
- Automatic page write mode disabled
- Number of FLASH wait states left unchanged

Table 11-4. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function nvm_set_config()

Sets the up the NVM hardware module based on the configuration.

```
enum status_code nvm_set_config(
    const struct nvm_config *const config)
```

Writes a given configuration of an NVM controller configuration to the hardware module, and initializes the internal device struct.

Table 11-5. Parameters

Data direction	Parameter name	Description
[in]	config	Configuration settings for the NVM controller

Note: The security bit must be cleared in order successfully use this function. This can only be done by a chip erase.

Returns

Status of the configuration procedure.

Table 11-6. Return Values

Return value	Description
STATUS_OK	If the initialization was a success
STATUS_BUSY	If the module was busy when the operation was attempted
STATUS_ERR_IO	If the security bit has been set, preventing the EEPROM and/or auxiliary space configuration from being altered

Function nvm_is_ready()

Checks if the NVM controller is ready to accept a new command.

```
bool nvm_is_ready( void )
```

Checks the NVM controller to determine if it is currently busy execution an operation, or ready for a new command.

Returns

Busy state of the NVM controller.

Table 11-7. Return Values

Return value	Description
true	If the hardware module is ready for a new command
false	If the hardware module is busy executing a command

11.6.3.2. NVM Access Management

Function `nvm_get_parameters()`

Reads the parameters of the NVM controller.

```
void nvm_get_parameters(  
    struct nvm_parameters *const parameters)
```

Retrieves the page size, number of pages, and other configuration settings of the NVM region.

Table 11-8. Parameters

Data direction	Parameter name	Description
[out]	parameters	Parameter structure, which holds page size and number of pages in the NVM memory

Function `nvm_write_buffer()`

Writes a number of bytes to a page in the NVM memory region.

```
enum status_code nvm_write_buffer(  
    const uint32_t destination_address,  
    const uint8_t * buffer,  
    uint16_t length)
```

Writes from a buffer to a given page address in the NVM memory.

Table 11-9. Parameters

Data direction	Parameter name	Description
[in]	destination_address	Destination page address to write to
[in]	buffer	Pointer to buffer where the data to write is stored
[in]	length	Number of bytes in the page to write

Note: If writing to a page that has previously been written to, the page's row should be erased (via `nvm_erase_row()`) before attempting to write new data to the page.

Note: For SAM D21 RWW devices, see `SAMD21_64K`, command `NVM_COMMAND_RWWE_WRITE_PAGE` must be executed before any other commands after writing a page, refer to errata 13588.

Note: If manual write mode is enabled, the write command must be executed after this function, otherwise the data will not write to NVM from page buffer.

Returns

Status of the attempt to write a page.

Table 11-10. Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read
STATUS_BUSY	NVM controller was busy when the operation was attempted

Return value	Description
STATUS_ERR_BAD_ADDRESS	The requested address was outside the acceptable range of the NVM memory region or not aligned to the start of a page
STATUS_ERR_INVALID_ARG	The supplied write length was invalid

Function `nvm_read_buffer()`

Reads a number of bytes from a page in the NVM memory region.

```
enum status_code nvm_read_buffer(
    const uint32_t source_address,
    uint8_t *const buffer,
    uint16_t length)
```

Reads a given number of bytes from a given page address in the NVM memory space into a buffer.

Table 11-11. Parameters

Data direction	Parameter name	Description
[in]	source_address	Source page address to read from
[out]	buffer	Pointer to a buffer where the content of the read page will be stored
[in]	length	Number of bytes in the page to read

Returns

Status of the page read attempt.

Table 11-12. Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read
STATUS_BUSY	NVM controller was busy when the operation was attempted
STATUS_ERR_BAD_ADDRESS	The requested address was outside the acceptable range of the NVM memory region or not aligned to the start of a page
STATUS_ERR_INVALID_ARG	The supplied read length was invalid

Function `nvm_update_buffer()`

Updates an arbitrary section of a page with new data.

```
enum status_code nvm_update_buffer(
    const uint32_t destination_address,
    uint8_t *const buffer,
    uint16_t offset,
    uint16_t length)
```

Writes from a buffer to a given page in the NVM memory, retaining any unmodified data already stored in the page.

Note: If manual write mode is enable, the write command must be executed after this function, otherwise the data will not write to NVM from page buffer.



Warning: This routine is unsafe if data integrity is critical; a system reset during the update process will result in up to one row of data being lost. If corruption must be avoided in all circumstances (including power loss or system reset) this function should not be used.

Table 11-13. Parameters

Data direction	Parameter name	Description
[in]	destination_address	Destination page address to write to
[in]	buffer	Pointer to buffer where the data to write is stored
[in]	offset	Number of bytes to offset the data write in the page
[in]	length	Number of bytes in the page to update

Returns

Status of the attempt to update a page.

Table 11-14. Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read
STATUS_BUSY	NVM controller was busy when the operation was attempted
STATUS_ERR_BAD_ADDRESS	The requested address was outside the acceptable range of the NVM memory region
STATUS_ERR_INVALID_ARG	The supplied length and offset was invalid

Function `nvm_erase_row()`

Erases a row in the NVM memory space.

```
enum status_code nvm_erase_row(  
    const uint32_t row_address)
```

Erases a given row in the NVM memory region.

Table 11-15. Parameters

Data direction	Parameter name	Description
[in]	row_address	Address of the row to erase

Returns

Status of the NVM row erase attempt.

Table 11-16. Return Values

Return value	Description
STATUS_OK	Requested NVM memory row was successfully erased
STATUS_BUSY	NVM controller was busy when the operation was attempted
STATUS_ERR_BAD_ADDRESS	The requested row address was outside the acceptable range of the NVM memory region or not aligned to the start of a row
STATUS_ABORTED	NVM erased error

Function `nvm_execute_command()`

Executes a command on the NVM controller.

```
enum status_code nvm_execute_command(
    const enum nvm_command command,
    const uint32_t address,
    const uint32_t parameter)
```

Executes an asynchronous command on the NVM controller, to perform a requested action such as an NVM page read or write operation.

Note: The function will return before the execution of the given command is completed.

Table 11-17. Parameters

Data direction	Parameter name	Description
[in]	command	Command to issue to the NVM controller
[in]	address	Address to pass to the NVM controller in NVM memory space
[in]	parameter	Parameter to pass to the NVM controller, not used for this driver

Returns

Status of the attempt to execute a command.

Table 11-18. Return Values

Return value	Description
STATUS_OK	If the command was accepted and execution is now in progress
STATUS_BUSY	If the NVM controller was already busy executing a command when the new command was issued
STATUS_ERR_IO	If the command was invalid due to memory or security locking
STATUS_ERR_INVALID_ARG	If the given command was invalid or unsupported
STATUS_ERR_BAD_ADDRESS	If the given address was invalid

Function `nvm_get_fuses()`

Get fuses from user row.

```
enum status_code nvm_get_fuses(  
    struct nvm_fusebits * fusebits)
```

Read out the fuse settings from the user row.

Table 11-19. Parameters

Data direction	Parameter name	Description
[in]	fusebits	Pointer to a 64-bit wide memory buffer of type struct <code>nvm_fusebits</code>

Returns

Status of read fuses attempt.

Table 11-20. Return Values

Return value	Description
STATUS_OK	This function will always return STATUS_OK

Function `nvm_set_fuses()`

Set fuses from user row.

```
enum status_code nvm_set_fuses(  
    struct nvm_fusebits * fb)
```

Set fuse settings from the user row.

Note: When writing to the user row, the values do not get loaded by the other modules on the device until a device reset occurs.

Table 11-21. Parameters

Data direction	Parameter name	Description
[in]	fusebits	Pointer to a 64-bit wide memory buffer of type struct <code>nvm_fusebits</code>

Returns

Status of read fuses attempt.

Table 11-22. Return Values

Return value	Description
STATUS_OK	This function will always return STATUS_OK
STATUS_BUSY	If the NVM controller was already busy executing a command when the new command was issued
STATUS_ERR_IO	If the command was invalid due to memory or security locking
STATUS_ERR_INVALID_ARG	If the given command was invalid or unsupported
STATUS_ERR_BAD_ADDRESS	If the given address was invalid

Function `nvm_is_page_locked()`

Checks whether the page region is locked.

```
bool nvm_is_page_locked(  
    uint16_t page_number)
```

Extracts the region to which the given page belongs and checks whether that region is locked.

Table 11-23. Parameters

Data direction	Parameter name	Description
[in]	page_number	Page number to be checked

Returns

Page lock status.

Table 11-24. Return Values

Return value	Description
true	Page is locked
false	Page is not locked

Function `nvm_get_error()`

Retrieves the error code of the last issued NVM operation.

```
enum nvm_error nvm_get_error( void )
```

Retrieves the error code from the last executed NVM operation. Once retrieved, any error state flags in the controller are cleared.

Note: The `nvm_is_ready()` function is an exception. Thus, errors retrieved after running this function should be valid for the function executed before `nvm_is_ready()`.

Returns

Error caused by the last NVM operation.

Table 11-25. Return Values

Return value	Description
NVM_ERROR_NONE	No error occurred in the last NVM operation
NVM_ERROR_LOCK	The last NVM operation attempted to access a locked region
NVM_ERROR_PROG	An invalid NVM command was issued

11.6.4. Enumeration Definitions

11.6.4.1. Enum `nvm_bod12_action`

What action should be triggered when BOD12 is detected.

Table 11-26. Members

Enum value	Description
NVM_BOD12_ACTION_NONE	No action
NVM_BOD12_ACTION_RESET	The BOD12 generates a reset
NVM_BOD12_ACTION_INTERRUPT	The BOD12 generates an interrupt

11.6.4.2. Enum nvm_bod33_action

What action should be triggered when BOD33 is detected.

Table 11-27. Members

Enum value	Description
NVM_BOD33_ACTION_NONE	No action
NVM_BOD33_ACTION_RESET	The BOD33 generates a reset
NVM_BOD33_ACTION_INTERRUPT	The BOD33 generates an interrupt

11.6.4.3. Enum nvm_bootloader_size

Available bootloader protection sizes in kilobytes.

Table 11-28. Members

Enum value	Description
NVM_BOOTLOADER_SIZE_128	Boot Loader Size is 32768 bytes
NVM_BOOTLOADER_SIZE_64	Boot Loader Size is 16384 bytes
NVM_BOOTLOADER_SIZE_32	Boot Loader Size is 8192 bytes
NVM_BOOTLOADER_SIZE_16	Boot Loader Size is 4096 bytes
NVM_BOOTLOADER_SIZE_8	Boot Loader Size is 2048 bytes
NVM_BOOTLOADER_SIZE_4	Boot Loader Size is 1024 bytes
NVM_BOOTLOADER_SIZE_2	Boot Loader Size is 512 bytes
NVM_BOOTLOADER_SIZE_0	Boot Loader Size is 0 bytes

11.6.4.4. Enum nvm_cache_readmode

Control how the NVM cache prefetch data from flash.

Table 11-29. Members

Enum value	Description
NVM_CACHE_READMODE_NO_MISS_PENALTY	The NVM Controller (cache system) does not insert wait states on a cache miss. Gives the best system performance.
NVM_CACHE_READMODE_LOW_POWER	Reduces power consumption of the cache system, but inserts a wait state each time there is a cache miss
NVM_CACHE_READMODE_DETERMINISTIC	The cache system ensures that a cache hit or miss takes the same amount of time, determined by the number of programmed flash wait states

11.6.4.5. Enum nvm_command

Table 11-30. Members

Enum value	Description
NVM_COMMAND_ERASE_ROW	Erases the addressed memory row
NVM_COMMAND_WRITE_PAGE	Write the contents of the page buffer to the addressed memory page
NVM_COMMAND_ERASE_AUX_ROW	Erases the addressed auxiliary memory row. Note: This command can only be given when the security bit is not set.
NVM_COMMAND_WRITE_AUX_ROW	Write the contents of the page buffer to the addressed auxiliary memory row. Note: This command can only be given when the security bit is not set.
NVM_COMMAND_LOCK_REGION	Locks the addressed memory region, preventing further modifications until the region is unlocked or the device is erased
NVM_COMMAND_UNLOCK_REGION	Unlocks the addressed memory region, allowing the region contents to be modified
NVM_COMMAND_PAGE_BUFFER_CLEAR	Clears the page buffer of the NVM controller, resetting the contents to all zero values
NVM_COMMAND_SET_SECURITY_BIT	Sets the device security bit, disallowing the changing of lock bits and auxiliary row data until a chip erase has been performed
NVM_COMMAND_ENTER_LOW_POWER_MODE	Enter power reduction mode in the NVM controller to reduce the power consumption of the system

Enum value	Description
NVM_COMMAND_EXIT_LOW_POWER_MODE	Exit power reduction mode in the NVM controller to allow other NVM commands to be issued
NVM_COMMAND_RWWEE_ERASE_ROW	Read while write (RWW) EEPROM area erase row
NVM_COMMAND_RWWEE_WRITE_PAGE	RWW EEPROM write page

11.6.4.6. Enum nvm_eeprom_emulator_size

Available space in flash dedicated for EEPROM emulator in bytes.

Table 11-31. Members

Enum value	Description
NVM_EEPROM_EMULATOR_SIZE_16384	EEPROM Size for EEPROM emulation is 16384 bytes
NVM_EEPROM_EMULATOR_SIZE_8192	EEPROM Size for EEPROM emulation is 8192 bytes
NVM_EEPROM_EMULATOR_SIZE_4096	EEPROM Size for EEPROM emulation is 4096 bytes
NVM_EEPROM_EMULATOR_SIZE_2048	EEPROM Size for EEPROM emulation is 2048 bytes
NVM_EEPROM_EMULATOR_SIZE_1024	EEPROM Size for EEPROM emulation is 1024 bytes
NVM_EEPROM_EMULATOR_SIZE_512	EEPROM Size for EEPROM emulation is 512 bytes
NVM_EEPROM_EMULATOR_SIZE_256	EEPROM Size for EEPROM emulation is 256 bytes
NVM_EEPROM_EMULATOR_SIZE_0	EEPROM Size for EEPROM emulation is 0 bytes

11.6.4.7. Enum nvm_error

Possible NVM controller error codes, which can be returned by the NVM controller after a command is issued.

Table 11-32. Members

Enum value	Description
NVM_ERROR_NONE	No errors
NVM_ERROR_LOCK	Lock error, a locked region was attempted accessed
NVM_ERROR_PROG	Program error, invalid command was executed

11.6.4.8. Enum nvm_sleep_power_mode

Power reduction modes of the NVM controller, to conserve power while the device is in sleep.

Table 11-33. Members

Enum value	Description
NVM_SLEEP_POWER_MODE_WAKEONACCESS	NVM controller exits low-power mode on first access after sleep
NVM_SLEEP_POWER_MODE_WAKEUPINSTANT	NVM controller exits low-power mode when the device exits sleep mode
NVM_SLEEP_POWER_MODE_ALWAYS_AWAKE	Power reduction mode in the NVM controller disabled

11.6.4.9. Enum `nvm_wdt_early_warning_offset`

This setting determine how many GCLK_WDT cycles before a watchdog time-out period an early warning interrupt should be triggered.

Table 11-34. Members

Enum value	Description
NVM_WDT_EARLY_WARNING_OFFSET_8	8 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_16	16 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_32	32 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_64	64 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_128	128 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_256	256 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_512	512 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_1024	1024 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_2048	2048 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_4096	4096 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_8192	8192 clock cycles
NVM_WDT_EARLY_WARNING_OFFSET_16384	16384 clock cycles

11.6.4.10. Enum `nvm_wdt_window_timeout`

Window mode time-out period in clock cycles.

Table 11-35. Members

Enum value	Description
NVM_WDT_WINDOW_TIMEOUT_PERIOD_8	8 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_16	16 clock cycles

Enum value	Description
NVM_WDT_WINDOW_TIMEOUT_PERIOD_32	32 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_64	64 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_128	128 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_256	256 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_512	512 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_1024	1024 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_2048	2048 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_4096	4096 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_8192	8192 clock cycles
NVM_WDT_WINDOW_TIMEOUT_PERIOD_16384	16384 clock cycles

11.7. Extra Information for NVM Driver

11.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
NVM	Non-Volatile Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory

11.7.2. Dependencies

This driver has the following dependencies:

- None

11.7.3. Errata

There are no errata related to this driver.

11.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Removed BOD12 reference, removed nvm_set_fuses() API
Added functions to read/write fuse settings

Changelog

Added support for NVM cache configuration

Updated initialization function to also enable the digital interface clock to the module if it is disabled

Initial Release

11.8. Examples for NVM Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Non-Volatile Memory \(NVM\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for NVM - Basic](#)

11.8.1. Quick Start Guide for NVM - Basic

In this use case, the NVM module is configured for:

- Power reduction mode enabled after sleep mode until first NVM access
- Automatic page write commands issued to commit data as pages are written to the internal buffer
- Zero wait states when reading FLASH memory
- No memory space for the EEPROM
- No protected bootloader section

This use case sets up the NVM controller to write a page of data to flash, and then read it back into the same buffer.

11.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_nvm(void)
{
    struct nvm_config config_nvm;

    nvm_get_config_defaults(&config_nvm);

    config_nvm.manual_page_write = false;

    nvm_set_config(&config_nvm);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_nvm();
```

Workflow

1. Create an NVM module configuration struct, which can be filled out to adjust the configuration of the NVM controller.

```
struct nvm_config config_nvm;
```

2. Initialize the NVM configuration struct with the module's default values.

```
nvm_get_config_defaults(&config_nvm);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Enable automatic page write mode. The new data will be written to NVM automatically.

```
config_nvm.manual_page_write = false;
```

Note: If automatic page write mode is disabled, the data will not write to NVM until the NVM write command has been invoked. For safe use of the NVM module, disable automatic page write mode and use write command to commit data is recommended.

4. Configure NVM controller with the created configuration struct settings.

```
nvm_set_config(&config_nvm);
```

11.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
uint8_t page_buffer[NVMCTRL_PAGE_SIZE];

for (uint32_t i = 0; i < NVMCTRL_PAGE_SIZE; i++) {
    page_buffer[i] = i;
}

enum status_code error_code;

do
{
    error_code = nvm_erase_row(
        100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);

do
{
    error_code = nvm_write_buffer(
        100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
        page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);

do
{
    error_code = nvm_read_buffer(
        100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
        page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);
```

Workflow

1. Set up a buffer, one NVM page in size, to hold data to read or write into NVM memory.

```
uint8_t page_buffer[NVMCTRL_PAGE_SIZE];
```


2. Fill the buffer with a pattern of data.

```
for (uint32_t i = 0; i < NVMCTRL_PAGE_SIZE; i++) {  
    page_buffer[i] = i;  
}
```

3. Create a variable to hold the error status from the called NVM functions.

```
enum status_code error_code;
```

4. Erase a page of NVM data. As the NVM could be busy initializing or completing a previous operation, a loop is used to retry the command while the NVM controller is busy.

```
do  
{  
    error_code = nvm_erase_row(  
        100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE);  
} while (error_code == STATUS_BUSY);
```

Note: This must be performed before writing new data into an NVM page.

5. Write the data buffer to the previously erased page of the NVM.

```
do  
{  
    error_code = nvm_write_buffer(  
        100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,  
        page_buffer, NVMCTRL_PAGE_SIZE);  
} while (error_code == STATUS_BUSY);
```

Note: The new data will be written to NVM memory automatically, as the NVM controller is configured in automatic page write mode.

6. Read back the written page of page from the NVM into the buffer.

```
do  
{  
    error_code = nvm_read_buffer(  
        100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,  
        page_buffer, NVMCTRL_PAGE_SIZE);  
} while (error_code == STATUS_BUSY);
```

12. SAM Peripheral Access Controller (PAC) Driver

This driver for Atmel® | SMART ARM®-based microcontroller provides an interface for the locking and unlocking of peripheral registers within the device. When a peripheral is locked, accidental writes to the peripheral will be blocked and a CPU exception will be raised.

The following peripherals are used by this module:

- PAC (Peripheral Access Controller)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

12.1. Prerequisites

There are no prerequisites for this module.

12.2. Module Overview

The SAM devices are fitted with a Peripheral Access Controller (PAC) that can be used to lock and unlock write access to a peripheral's registers (see [Non-Writable Registers](#)). Locking a peripheral minimizes the risk of unintended configuration changes to a peripheral as a consequence of [Run-away Code](#) or use of a [Faulty Module Pointer](#).

Physically, the PAC restricts write access through the AHB bus to registers used by the peripheral, making the register non-writable. PAC locking of modules should be implemented in configuration critical applications where avoiding unintended peripheral configuration changes are to be regarded in the highest of priorities.

All interrupt must be disabled while a peripheral is unlocked to make sure correct lock/unlock scheme is upheld.

12.2.1. Locking Scheme

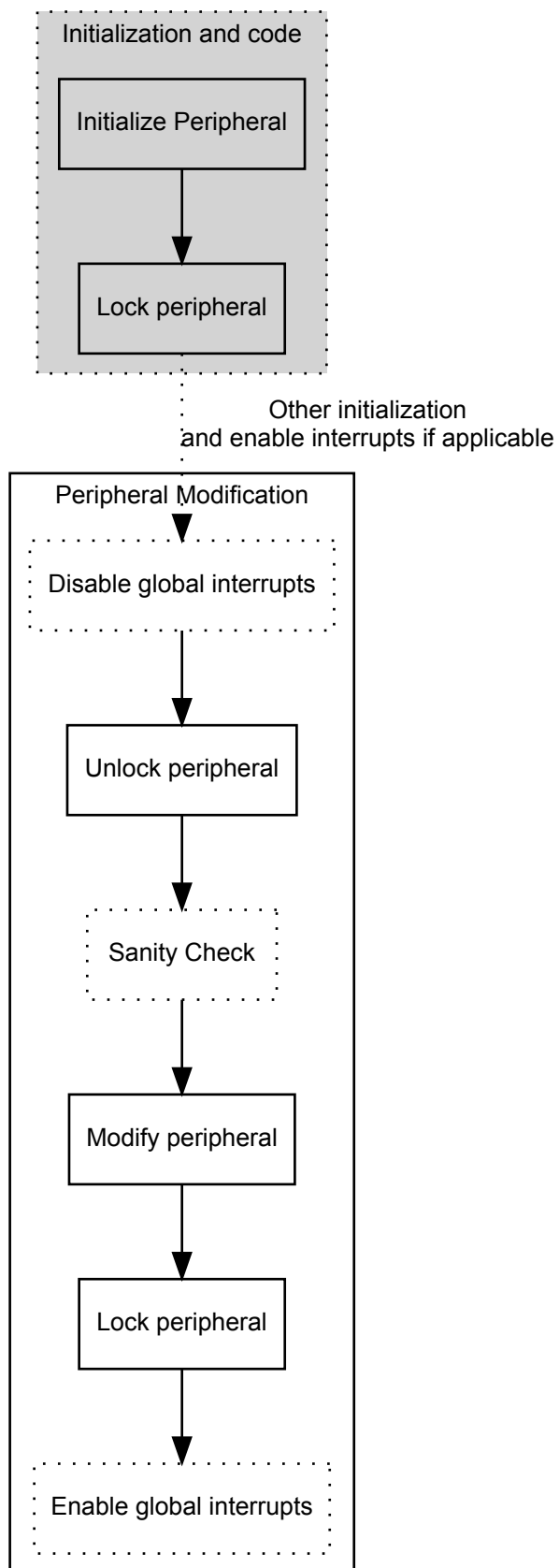
The module has a built in safety feature requiring that an already locked peripheral is not relocked, and that already unlocked peripherals are not unlocked again. Attempting to unlock and already unlocked peripheral, or attempting to lock a peripheral that is currently locked will generate a CPU exception. This implies that the implementer must keep strict control over the peripheral's lock-state before modifying

them. With this added safety, the probability of stopping runaway code increases as the program pointer can be caught inside the exception handler, and necessary countermeasures can be initiated. The implementer should also consider using sanity checks after an unlock has been performed to further increase the security.

12.2.2. Recommended Implementation

A recommended implementation of the PAC can be seen in [Figure 12-1](#).

Figure 12-1. Recommended Implementation



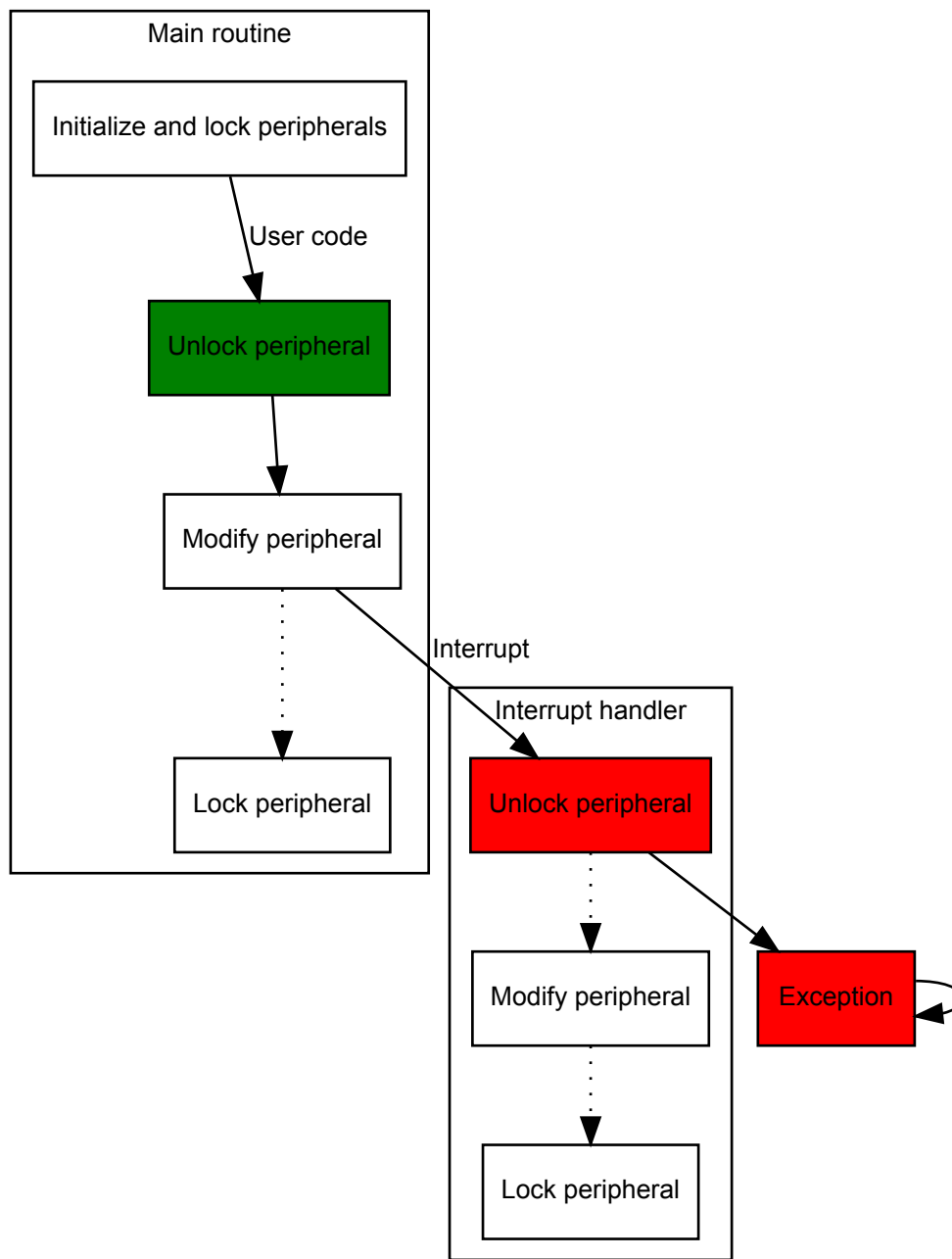
12.2.3. Why Disable Interrupts

Global interrupts must be disabled while a peripheral is unlocked as an interrupt handler would not know the current state of the peripheral lock. If the interrupt tries to alter the lock state, it can cause an exception as it potentially tries to unlock an already unlocked peripheral. Reading current lock state is to be avoided as it removes the security provided by the PAC ([Reading Lock State](#)).

Note: Global interrupts should also be disabled when a peripheral is unlocked inside an interrupt handler.

An example to illustrate the potential hazard of not disabling interrupts is shown in [Figure 12-2](#).

Figure 12-2. Why Disable Interrupts



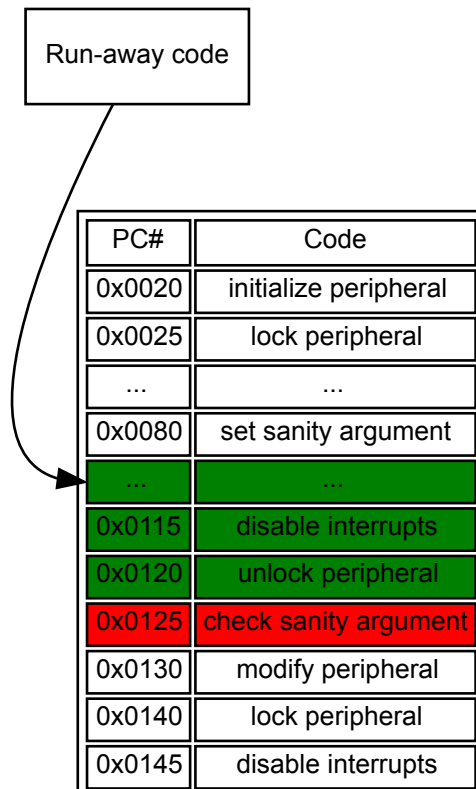
12.2.4. Run-away Code

Run-away code can be caused by the MCU being operated outside its specification, faulty code, or EMI issues. If a runaway code occurs, it is favorable to catch the issue as soon as possible. With a correct implementation of the PAC, the runaway code can potentially be stopped.

A graphical example showing how a PAC implementation will behave for different circumstances of runaway code is shown in [Figure 12-3](#) and [Figure 12-4](#).

Figure 12-3. Run-away Code

1. Run-away code is caught in sanity check.
A CPU exception is executed.



2. Run-away code is caught when modifying locked peripheral. A CPU exception is executed.

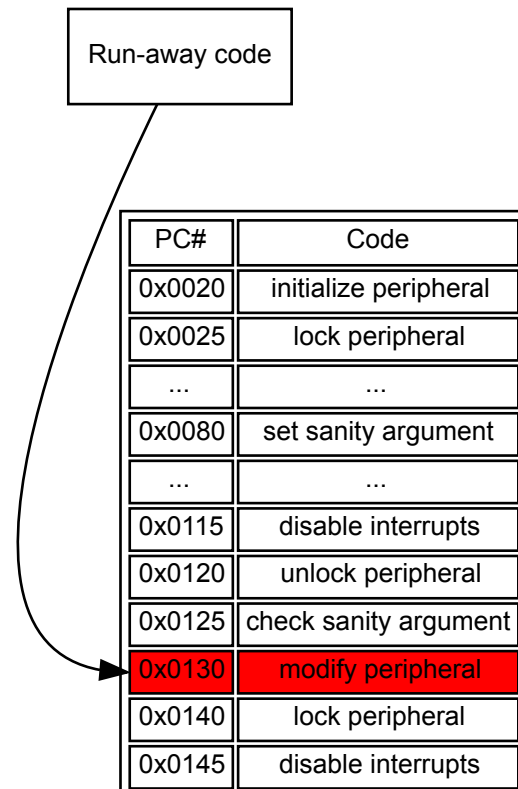
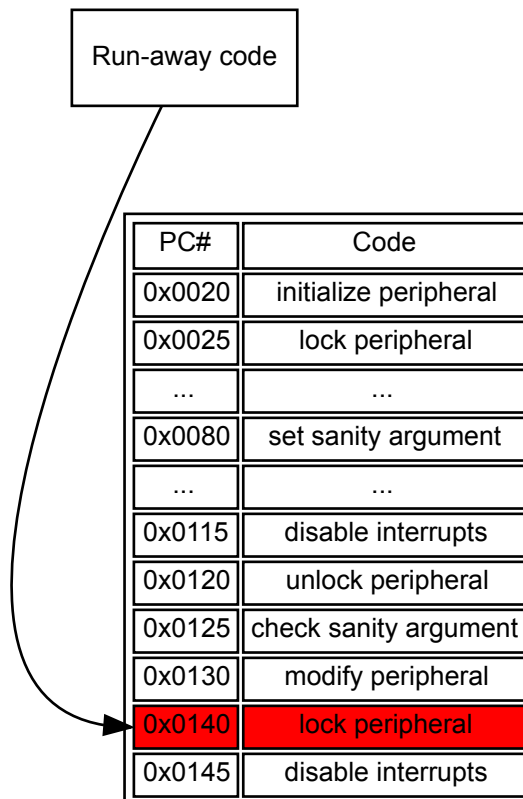
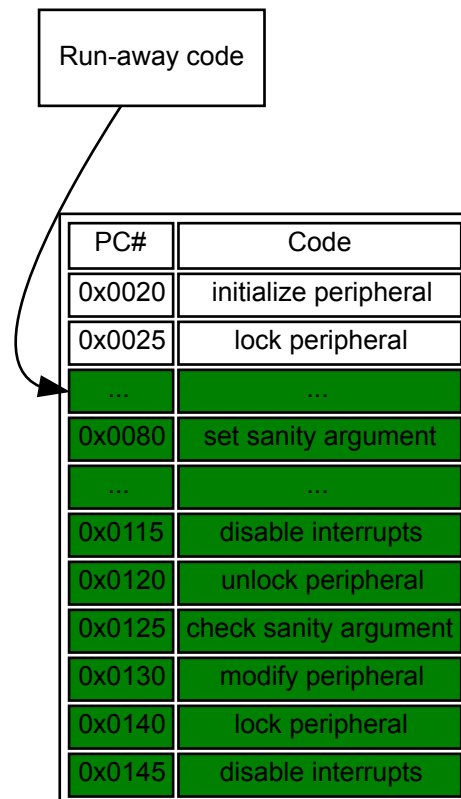


Figure 12-4. Run-away Code

3. Run-away code is caught when locking locked peripheral. A CPU exception is executed.



4. Run-away code is not caught.



In the example, green indicates that the command is allowed, red indicates where the runaway code will be caught, and the arrow where the runaway code enters the application. In special circumstances, like example 4 above, the runaway code will not be caught. However, the protection scheme will greatly enhance peripheral configuration security from being affected by runaway code.

12.2.4.1. Key-Argument

To protect the module functions against runaway code themselves, a key is required as one of the input arguments. The key-argument will make sure that runaway code entering the function without a function call will be rejected before inflicting any damage. The argument is simply set to be the bitwise inverse of the module flag, i.e.

```
system_peripheral_<lock_state>(SYSTEM_PERIPHERAL_<module>,
    ~SYSTEM_PERIPHERAL_<module>);
```

Where the lock state can be either lock or unlock, and module refer to the peripheral that is to be locked/unlocked.

12.2.5. Faulty Module Pointer

The PAC also protects the application from user errors such as the use of incorrect module pointers in function arguments, given that the module is locked. It is therefore recommended that any unused peripheral is locked during application initialization.

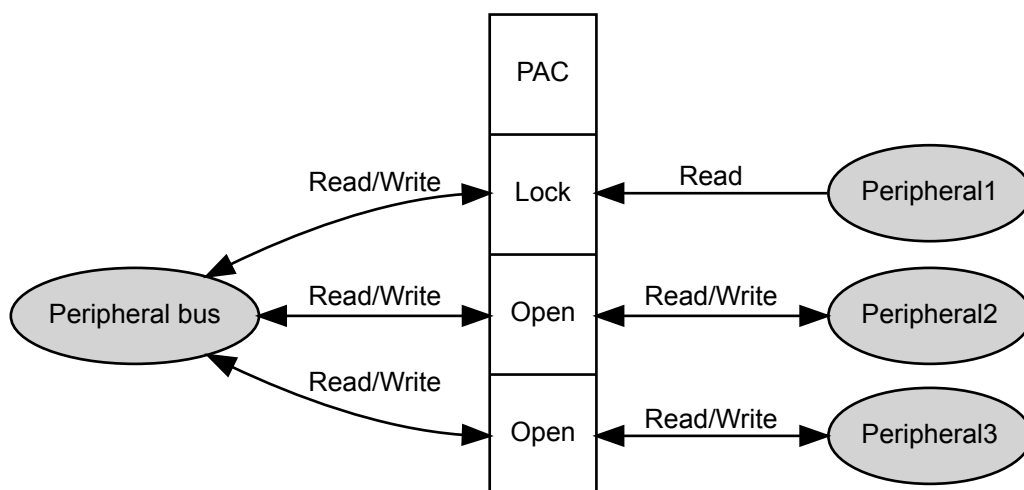
12.2.6. Use of `__no_inline`

Using the function attribute `__no_inline` will ensure that there will only be one copy of each functions in the PAC driver API in the application. This will lower the likelihood that runaway code will hit any of these functions.

12.2.7. Physical Connection

Figure 12-5 shows how this module is interconnected within the device.

Figure 12-5. Physical Connection



12.3. Special Considerations

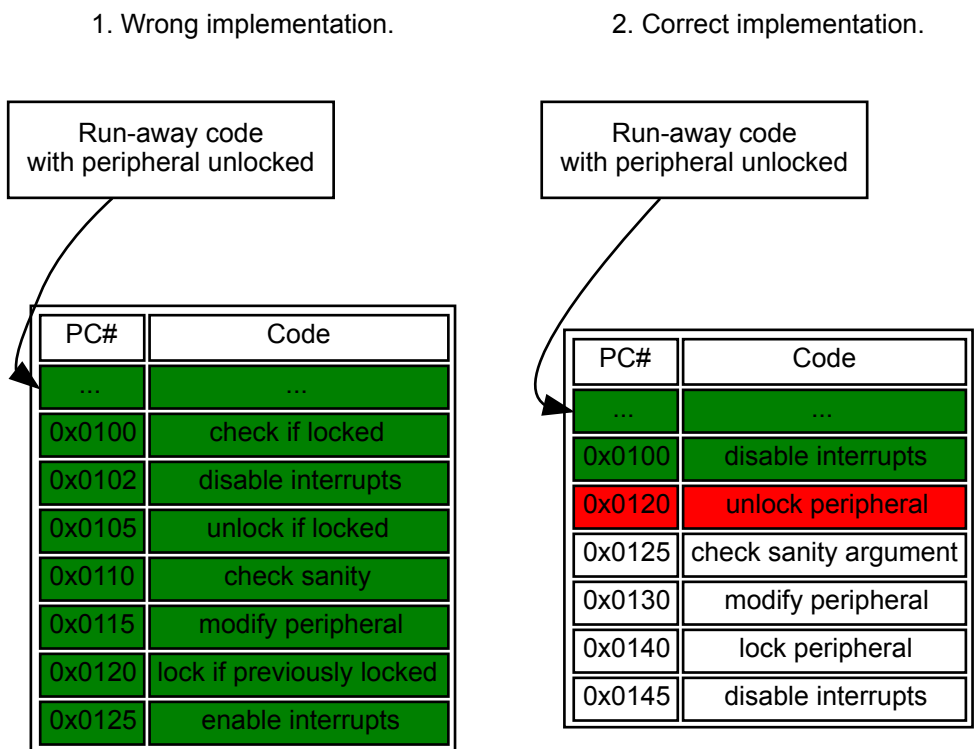
12.3.1. Non-Writable Registers

Not all registers in a given peripheral can be set non-writable. Which registers this applies to is showed in [List of Non-Write Protected Registers](#) and the peripheral's subsection "Register Access Protection" in the device datasheet.

12.3.2. Reading Lock State

Reading the state of the peripheral lock is to be avoided as it greatly compromises the protection initially provided by the PAC. If a lock/unlock is implemented conditionally, there is a risk that eventual errors are not caught in the protection scheme. Examples indicating the issue are shown in [Figure 12-6](#).

Figure 12-6. Reading Lock State



In the left figure above, one can see the runaway code continues as all illegal operations are conditional. On the right side figure, the runaway code is caught as it tries to unlock the peripheral.

12.4. Extra Information

For extra information, see [Extra Information for PAC Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

12.5. Examples

For a list of examples related to this driver, see [Examples for PAC Driver](#).

12.6. API Overview

12.6.1. Macro Definitions

12.6.1.1. Macro SYSTEM_PERIPHERAL_ID

```
#define SYSTEM_PERIPHERAL_ID(peripheral)
```

Retrieves the ID of a specified peripheral name, giving its peripheral bus location.

Table 12-1. Parameters

Data direction	Parameter name	Description
[in]	peripheral	Name of the peripheral instance

12.6.2. Function Definitions

12.6.2.1. Peripheral Lock and Unlock

Function `system_peripheral_lock()`

Lock a given peripheral's control registers.

```
__no_inline enum status_code system_peripheral_lock(  
    const uint32_t peripheral_id,  
    const uint32_t key)
```

Locks a given peripheral's control registers, to deny write access to the peripheral to prevent accidental changes to the module's configuration.



Warning: Locking an already locked peripheral will cause a CPU exception, and terminate program execution.

Table 12-2. Parameters

Data direction	Parameter name	Description
[in]	peripheral_id	ID for the peripheral to be locked, sourced via the SYSTEM_PERIPHERAL_ID macro
[in]	key	Bitwise inverse of peripheral ID, used as key to reduce the chance of accidental locking. See Key-Argument

Returns

Status of the peripheral lock procedure.

Table 12-3. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied

Function `system_peripheral_unlock()`

Unlock a given peripheral's control registers.

```
__no_inline enum status_code system_peripheral_unlock(  
    const uint32_t peripheral_id,  
    const uint32_t key)
```

Unlocks a given peripheral's control registers, allowing write access to the peripheral so that changes can be made to the module's configuration.



Warning: Unlocking an already locked peripheral will cause a CUP exception, and terminate program execution.

Table 12-4. Parameters

Data direction	Parameter name	Description
[in]	peripheral_id	ID for the peripheral to be unlocked, sourced via the SYSTEM_PERIPHERAL_ID macro
[in]	key	Bitwise inverse of peripheral ID, used as key to reduce the chance of accidental unlocking. See Key-Argument

Returns

Status of the peripheral unlock procedure.

Table 12-5. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied

12.6.2.2. APIs available for SAM L21/L22/C20/C21.

Function `system_peripheral_lock_always()`

Lock a given peripheral's control registers until hardware reset.

```
__no_inline enum status_code system_peripheral_lock_always(  
    const uint32_t peripheral_id,  
    const uint32_t key)
```

Locks a given peripheral's control registers, to deny write access to the peripheral to prevent accidental changes to the module's configuration. After lock, the only way to unlock is hardware reset.



Warning: Locking an already locked peripheral will cause a CPU exception, and terminate program execution.

Table 12-6. Parameters

Data direction	Parameter name	Description
[in]	peripheral_id	ID for the peripheral to be locked, sourced via the SYSTEM_PERIPHERAL_ID macro
[in]	key	Bitwise inverse of peripheral ID, used as key to reduce the chance of accidental locking. See Key-Argument

Returns

Status of the peripheral lock procedure.

Table 12-7. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied

Function system_pac_enable_interrupt()

Enable PAC interrupt.

```
void system_pac_enable_interrupt( void )
```

Enable PAC interrupt so can trigger execution on peripheral access error, see SYSTEM_Handler().

Function system_pac_disable_interrupt()

Disable PAC interrupt.

```
void system_pac_disable_interrupt( void )
```

Disable PAC interrupt on peripheral access error.

Function system_pac_enable_event()

Enable PAC event output.

```
void system_pac_enable_event( void )
```

Enable PAC event output on peripheral access error.

Function system_pac_disable_event()

Disable PAC event output.

```
void system_pac_disable_event( void )
```

Disable PAC event output on peripheral access error.

12.7. Extra Information for PAC Driver

12.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
AC	Analog Comparator
ADC	Analog-to-Digital Converter
EVSYS	Event System
NMI	Non-Maskable Interrupt
NVMCTRL	Non-Volatile Memory Controller

Acronym	Description
PAC	Peripheral Access Controller
PM	Power Manager
RTC	Real-Time Counter
SERCOM	Serial Communication Interface
SYSCTRL	System Controller
TC	Timer/Counter
WDT	Watch Dog Timer

12.7.2. Dependencies

This driver has the following dependencies:

- None

12.7.3. Errata

There are no errata related to this driver.

12.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

12.8. Examples for PAC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Peripheral Access Controller \(PAC\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for PAC - Basic](#)

12.8.1. Quick Start Guide for PAC - Basic

In this use case, the peripheral-lock will be used to lock and unlock the PORT peripheral access, and show how to implement the PAC module when the PORT registers needs to be altered. The PORT will be set up as follows:

- One pin in input mode, with pull-up and falling edge-detect
- One pin in output mode

12.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void config_port_pins(void)
{
    struct port_config pin_conf;

    port_get_config_defaults(&pin_conf);

    pin_conf.direction = PORT_PIN_DIR_INPUT;
    pin_conf.input_pull = PORT_PIN_PULL_UP;
    port_pin_set_config(BUTTON_0_PIN, &pin_conf);

    pin_conf.direction = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LED_0_PIN, &pin_conf);
}
```

Add to user application initialization (typically the start of `main()`):

```
config_port_pins();
```

12.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_init();

config_port_pins();

system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
    ~SYSTEM_PERIPHERAL_ID(PORT));

#if(SAML21) || (SAML22) || (SAMC21) || defined(__DOXYGEN__)
    system_pac_enable_interrupt();
#endif
    system_interrupt_enable_global();

while (port_pin_get_input_level(BUTTON_0_PIN)) {
    /* Wait for button press */
}

system_interrupt_enter_critical_section();

system_peripheral_unlock(SYSTEM_PERIPHERAL_ID(PORT),
    ~SYSTEM_PERIPHERAL_ID(PORT));

port_pin_toggle_output_level(LED_0_PIN);

system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
    ~SYSTEM_PERIPHERAL_ID(PORT));

system_interrupt_leave_critical_section();

while (1) {
    /* Do nothing */
}
```

Workflow

1. Configure some GPIO port pins for input and output.

```
config_port_pins();
```

2. Lock peripheral access for the PORT module; attempting to update the module while it is in a protected state will cause a CPU exception. For SAM D20/D21/D10/D11/R21/DA0/DA1, it is Hard Fault exception; For SAM L21/C21, it is system exception, see SYSTEM_Handler().

```
system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),  
    ~SYSTEM_PERIPHERAL_ID(PORT));
```

3. Enable global interrupts.

```
#if (SAML21) || (SAML22) || (SAMC21) || defined(__DOXYGEN__)  
    system_pac_enable_interrupt();  
#endif  
system_interrupt_enable_global();
```

4. Loop to wait for a button press before continuing.

```
while (port_pin_get_input_level(BUTTON_0_PIN)) {  
    /* Wait for button press */  
}
```

5. Enter a critical section, so that the PAC module can be unlocked safely and the peripheral manipulated without the possibility of an interrupt modifying the protected module's state.

```
system_interrupt_enter_critical_section();
```

6. Unlock the PORT peripheral registers.

```
system_peripheral_unlock(SYSTEM_PERIPHERAL_ID(PORT),  
    ~SYSTEM_PERIPHERAL_ID(PORT));
```

7. Toggle pin 11, and clear edge detect flag.

```
port_pin_toggle_output_level(LED_0_PIN);
```

8. Lock the PORT peripheral registers.

```
system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),  
    ~SYSTEM_PERIPHERAL_ID(PORT));
```

9. Exit the critical section to allow interrupts to function normally again.

```
system_interrupt_leave_critical_section();
```

10. Enter an infinite while loop once the module state has been modified successfully.

```
while (1) {  
    /* Do nothing */  
}
```

12.9. List of Non-Write Protected Registers

Look in device datasheet peripheral's subsection "Register Access Protection" to see which is actually available for your device.

Module	Non-write protected register
AC	INTFLAG
	STATUSA
	STATUSB
	STATUSC
ADC	INTFLAG
	STATUS
	RESULT
EVSYS	INTFLAG
	CHSTATUS
NVMCTRL	INTFLAG
	STATUS
PM	INTFLAG
PORT	N/A
RTC	INTFLAG
	READREQ
	STATUS
SYSCTRL	INTFLAG
SERCOM	INTFLAG
	STATUS
	DATA
TC	INTFLAG
	STATUS
WDT	INTFLAG

Module	Non-write protected register
	STATUS
	(CLEAR)

13. SAM Port (PORT) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's General Purpose Input/Output (GPIO) pin functionality, for manual pin state reading and writing.

The following peripheral is used by this module:

- PORT (GPIO Management)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

13.1. Prerequisites

There are no prerequisites for this module.

13.2. Module Overview

The device GPIO (PORT) module provides an interface between the user application logic and external hardware peripherals, when general pin state manipulation is required. This driver provides an easy-to-use interface to the physical pin input samplers and output drivers, so that pins can be read from or written to for general purpose external hardware control.

13.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_PORT_INPUT_EVENT	SAM L21/L22/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

13.2.2. Physical and Logical GPIO Pins

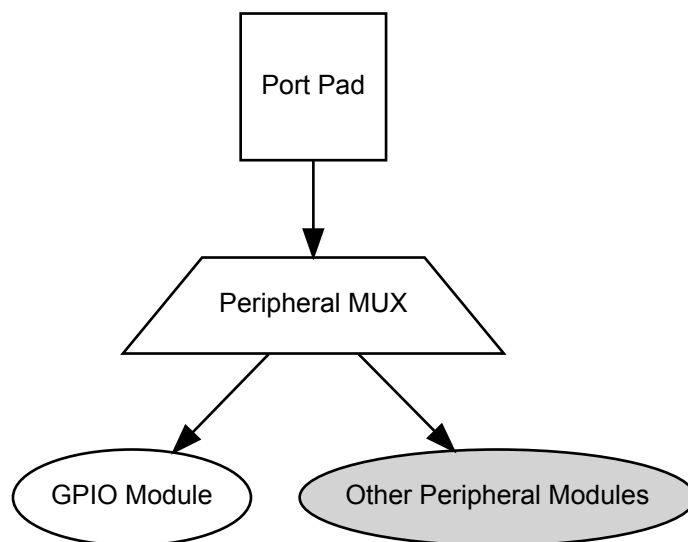
SAM devices use two naming conventions for the I/O pins in the device; one physical and one logical. Each physical pin on a device package is assigned both a physical port and pin identifier (e.g. "PORTA.

0") as well as a monotonically incrementing logical GPIO number (e.g. "GPIO0"). While the former is used to map physical pins to their physical internal device module counterparts, for simplicity the design of this driver uses the logical GPIO numbers instead.

13.2.3. Physical Connection

[Figure 13-1](#) shows how this module is interconnected within the device.

Figure 13-1. Physical Connection



13.3. Special Considerations

The SAM port pin input sampler can be disabled when the pin is configured in pure output mode to save power; reading the pin state of a pin configured in output-only mode will read the logical output state that was last set.

13.4. Extra Information

For extra information, see [Extra Information for PORT Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

13.5. Examples

For a list of examples related to this driver, see [Examples for PORT Driver](#).

13.6. API Overview

13.6.1. Structure Definitions

13.6.1.1. Struct port_config

Configuration structure for a port pin instance. This structure should be initialized by the [port_get_config_defaults\(\)](#) function before being modified by the user application.

Table 13-1. Members

Type	Name	Description
enum port_pin_dir	direction	Port buffer input/output direction
enum port_pin_pull	input_pull	Port pull-up/pull-down for input pins
bool	powersave	Enable lowest possible powerstate on the pin Note: All other configurations will be ignored, the pin will be disabled.

13.6.1.2. Struct port_input_event_config

Configuration structure for a port input event.

Table 13-2. Members

Type	Name	Description
enum port_input_event_action	action	Port input event action
uint8_t	gpio_pin	GPIO pin

13.6.2. Macro Definitions

13.6.2.1. Driver Feature Definition

Define port features set according to different device family.

Macro FEATURE_PORT_INPUT_EVENT

```
#define FEATURE_PORT_INPUT_EVENT
```

Event input control feature support for PORT group.

13.6.2.2. PORT Alias Macros

Macro PORTA

```
#define PORTA
```

Convenience definition for GPIO module group A on the device (if available).

Macro PORTB

```
#define PORTB
```

Convenience definition for GPIO module group B on the device (if available).

Macro PORTC

```
#define PORTC
```

Convenience definition for GPIO module group C on the device (if available).

Macro PORTD

```
#define PORTD
```

Convenience definition for GPIO module group D on the device (if available).

13.6.3. Function Definitions

13.6.3.1. State Reading/Writing (Physical Group Orientated)

Function `port_get_group_from_gpio_pin()`

Retrieves the PORT module group instance from a given GPIO pin number.

```
PortGroup * port_get_group_from_gpio_pin(  
    const uint8_t gpio_pin)
```

Retrieves the PORT module group instance associated with a given logical GPIO pin number.

Table 13-3. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to convert

Returns

Base address of the associated PORT module.

Function `port_group_get_input_level()`

Retrieves the state of a group of port pins that are configured as inputs.

```
uint32_t port_group_get_input_level(  
    const PortGroup *const port,  
    const uint32_t mask)
```

Reads the current logic level of a port module's pins and returns the current levels as a bitmask.

Table 13-4. Parameters

Data direction	Parameter name	Description
[in]	port	Base of the PORT module to read from
[in]	mask	Mask of the port pin(s) to read

Returns

Status of the port pin(s) input buffers.

Function `port_group_get_output_level()`

Retrieves the state of a group of port pins that are configured as outputs.

```
uint32_t port_group_get_output_level(  
    const PortGroup *const port,  
    const uint32_t mask)
```

Reads the current logical output level of a port module's pins and returns the current levels as a bitmask.

Table 13-5. Parameters

Data direction	Parameter name	Description
[in]	port	Base of the PORT module to read from
[in]	mask	Mask of the port pin(s) to read

Returns

Status of the port pin(s) output buffers.

Function `port_group_set_output_level()`

Sets the state of a group of port pins that are configured as outputs.

```
void port_group_set_output_level(  
    PortGroup *const port,  
    const uint32_t mask,  
    const uint32_t level_mask)
```

Sets the current output level of a port module's pins to a given logic level.

Table 13-6. Parameters

Data direction	Parameter name	Description
[out]	port	Base of the PORT module to write to
[in]	mask	Mask of the port pin(s) to change
[in]	level_mask	Mask of the port level(s) to set

Function `port_group_toggle_output_level()`

Toggles the state of a group of port pins that are configured as an outputs.

```
void port_group_toggle_output_level(  
    PortGroup *const port,  
    const uint32_t mask)
```

Toggles the current output levels of a port module's pins.

Table 13-7. Parameters

Data direction	Parameter name	Description
[out]	port	Base of the PORT module to write to
[in]	mask	Mask of the port pin(s) to toggle

13.6.3.2. Configuration and Initialization

Function `port_get_config_defaults()`

Initializes a Port pin/group configuration structure to defaults.

```
void port_get_config_defaults(  
    struct port_config *const config)
```

Initializes a given Port pin/group configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Input mode with internal pull-up enabled

Table 13-8. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `port_pin_set_config()`

Writes a Port pin configuration to the hardware module.

```
void port_pin_set_config(  
    const uint8_t gpio_pin,  
    const struct port_config *const config)
```

Writes out a given configuration of a Port pin configuration to the hardware module.

Note: If the pin direction is set as an output, the pull-up/pull-down input configuration setting is ignored.

Table 13-9. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure
[in]	config	Configuration settings for the pin

Function `port_group_set_config()`

Writes a Port group configuration group to the hardware module.

```
void port_group_set_config(  
    PortGroup *const port,  
    const uint32_t mask,  
    const struct port_config *const config)
```

Writes out a given configuration of a Port group configuration to the hardware module.

Note: If the pin direction is set as an output, the pull-up/pull-down input configuration setting is ignored.

Table 13-10. Parameters

Data direction	Parameter name	Description
[out]	port	Base of the PORT module to write to
[in]	mask	Mask of the port pin(s) to configure
[in]	config	Configuration settings for the pin group

13.6.3.3. State Reading/Writing (Logical Pin Orientated)

Function `port_pin_get_input_level()`

Retrieves the state of a port pin that is configured as an input.

```
bool port_pin_get_input_level(
    const uint8_t gpio_pin)
```

Reads the current logic level of a port pin and returns the current level as a Boolean value.

Table 13-11. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to read

Returns

Status of the port pin's input buffer.

Function `port_pin_get_output_level()`

Retrieves the state of a port pin that is configured as an output.

```
bool port_pin_get_output_level(
    const uint8_t gpio_pin)
```

Reads the current logical output level of a port pin and returns the current level as a Boolean value.

Table 13-12. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to read

Returns

Status of the port pin's output buffer.

Function `port_pin_set_output_level()`

Sets the state of a port pin that is configured as an output.

```
void port_pin_set_output_level(
    const uint8_t gpio_pin,
    const bool level)
```

Sets the current output level of a port pin to a given logic level.

Table 13-13. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to write to
[in]	level	Logical level to set the given pin to

Function port_pin_toggle_output_level()

Toggles the state of a port pin that is configured as an output.

```
void port_pin_toggle_output_level(
    const uint8_t gpio_pin)
```

Toggles the current output level of a port pin.

Table 13-14. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to toggle

13.6.3.4. Port Input Event**Function port_enable_input_event()**

Enable the port event input.

```
enum status_code port_enable_input_event(
    const uint8_t gpio_pin,
    const enum port_input_event n)
```

Enable the port event input with the given pin and event.

Table 13-15. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin
[in]	n	Port input event

Table 13-16. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid parameter
STATUS_OK	Successfully

Function port_disable_input_event()

Disable the port event input.

```
enum status_code port_disable_input_event(
    const uint8_t gpio_pin,
    const enum port_input_event n)
```

Disable the port event input with the given pin and event.

Table 13-17. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin
[in]	gpio_pin	Port input event

Table 13-18. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid parameter
STATUS_OK	Successfully

Function port_input_event_get_config_defaults()

Retrieve the default configuration for port input event.

```
void port_input_event_get_config_defaults(
    struct port_input_event_config *const config)
```

Fills a configuration structure with the default configuration for port input event:

- Event output to pin
- Event action to be executed on PIN 0

Table 13-19. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function port_input_event_set_config()

Configure port input event.

```
enum status_code port_input_event_set_config(
    const enum port_input_event n,
    struct port_input_event_config *const config)
```

Configures port input event with the given configuration settings.

Table 13-20. Parameters

Data direction	Parameter name	Description
[in]	config	Port input even configuration structure containing the new config

Table 13-21. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid parameter
STATUS_OK	Successfully

13.6.4. Enumeration Definitions

13.6.4.1. Enum port_input_event

List of port input events.

Table 13-22. Members

Enum value	Description
PORT_INPUT_EVENT_0	Port input event 0
PORT_INPUT_EVENT_1	Port input event 1
PORT_INPUT_EVENT_2	Port input event 2
PORT_INPUT_EVENT_3	Port input event 3

13.6.4.2. Enum port_input_event_action

List of port input events action on pin.

Table 13-23. Members

Enum value	Description
PORT_INPUT_EVENT_ACTION_OUT	Event out to pin
PORT_INPUT_EVENT_ACTION_SET	Set output register of pin on event
PORT_INPUT_EVENT_ACTION_CLR	Clear output register pin on event
PORT_INPUT_EVENT_ACTION_TGL	Toggle output register pin on event

13.6.4.3. Enum port_pin_dir

Enum for the possible pin direction settings of the port pin configuration structure, to indicate the direction the pin should use.

Table 13-24. Members

Enum value	Description
PORT_PIN_DIR_INPUT	The pin's input buffer should be enabled, so that the pin state can be read
PORT_PIN_DIR_OUTPUT	The pin's output buffer should be enabled, so that the pin state can be set
PORT_PIN_DIR_OUTPUT_WTH_READBACK	The pin's output and input buffers should be enabled, so that the pin state can be set and read back

13.6.4.4. Enum port_pin_pull

Enum for the possible pin pull settings of the port pin configuration structure, to indicate the type of logic level pull the pin should use.

Table 13-25. Members

Enum value	Description
PORT_PIN_PULL_NONE	No logical pull should be applied to the pin
PORT_PIN_PULL_UP	Pin should be pulled up when idle
PORT_PIN_PULL_DOWN	Pin should be pulled down when idle

13.7. Extra Information for PORT Driver

13.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
GPIO	General Purpose Input/Output
MUX	Multiplexer

13.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

13.7.3. Errata

There are no errata related to this driver.

13.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added input event feature
Initial release

13.8. Examples for PORT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Port \(PORT\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for PORT - Basic](#)

13.8.1. Quick Start Guide for PORT - Basic

In this use case, the PORT module is configured for:

- One pin in input mode, with pull-up enabled
- One pin in output mode

This use case sets up the PORT to read the current state of a GPIO pin set as an input, and mirrors the opposite logical state on a pin configured as an output.

13.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_port_pins(void)
{
    struct port_config config_port_pin;
    port_get_config_defaults(&config_port_pin);

    config_port_pin.direction = PORT_PIN_DIR_INPUT;
    config_port_pin.input_pull = PORT_PIN_PULL_UP;
    port_pin_set_config(BUTTON_0_PIN, &config_port_pin);

    config_port_pin.direction = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LED_0_PIN, &config_port_pin);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_port_pins();
```

Workflow

1. Create a PORT module pin configuration struct, which can be filled out to adjust the configuration of a single port pin.

```
struct port_config config_port_pin;
```

2. Initialize the pin configuration struct with the module's default values.

```
port_get_config_defaults(&config_port_pin);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to request an input pin.

```
config_port_pin.direction = PORT_PIN_DIR_INPUT;
config_port_pin.input_pull = PORT_PIN_PULL_UP;
```

4. Configure push button pin with the initialized pin configuration struct, to enable the input sampler on the pin.

```
port_pin_set_config(BUTTON_0_PIN, &config_port_pin);
```

5. Adjust the configuration struct to request an output pin.

```
config_port_pin.direction = PORT_PIN_DIR_OUTPUT;
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

6. Configure LED pin with the initialized pin configuration struct, to enable the output driver on the pin.

```
port_pin_set_config(LED_0_PIN, &config_port_pin);
```

13.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    bool pin_state = port_pin_get_input_level(BUTTON_0_PIN);  
    port_pin_set_output_level(LED_0_PIN, !pin_state);  
}
```

Workflow

1. Read in the current input sampler state of push button pin, which has been configured as an input in the use-case setup code.

```
bool pin_state = port_pin_get_input_level(BUTTON_0_PIN);
```

2. Write the inverted pin level state to LED pin, which has been configured as an output in the use-case setup code.

```
port_pin_set_output_level(LED_0_PIN, !pin_state);
```

14. SAM RTC Count (RTC COUNT) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Real Time Clock functionality in Count operating mode, for the configuration and retrieval of the current RTC counter value. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- RTC (Real Time Clock)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

14.1. Prerequisites

There are no prerequisites for this module.

14.2. Module Overview

The RTC module in the SAM devices is a 32-bit counter, with a 10-bit programmable prescaler. Typically, the RTC clock is run continuously, including in the device's low-power sleep modes, to track the current time and date information. The RTC can be used as a source to wake up the system at a scheduled time or periodically using the alarm functions.

In this driver, the RTC is operated in Count mode. This allows for an easy integration of an asynchronous counter into a user application, which is capable of operating while the device is in sleep mode.

Whilst operating in Count mode, the RTC features:

- 16-bit counter mode
 - Selectable counter period
 - Up to six configurable compare values
- 32-bit counter mode

- Clear counter value on match
- Up to four configurable compare values

14.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_RTC_PERIODIC_INT	SAM L21/L22/C20/C21/R30
FEATURE_RTC_PRESCALER_OFF	SAM L21/L22/C20/C21/R30
FEATURE_RTC_CLOCK_SELECTION	SAM L21/L22/C20/C21/R30
FEATURE_RTC_GENERAL_PURPOSE_REG	SAM L21/L22/R30
FEATURE_RTC_CONTINUOUSLY_UPDATED	SAM D20, SAM D21, SAM R21, SAM D10, SAM D11, SAM DA1
FEATURE_RTC_TAMPER_DETECTION	SAM L22

Note: The specific features are only available in the driver when the selected device supports those features.

14.3. Compare and Overflow

The RTC can be used with up to 4/6 compare values (depending on selected operation mode). These compare values will trigger on match with the current RTC counter value, and can be set up to trigger an interrupt, event, or both. The RTC can also be configured to clear the counter value on compare match in 32-bit mode, resetting the count value back to zero.

If the RTC is operated without the Clear on Match option enabled, or in 16-bit mode, the RTC counter value will instead be cleared on overflow once the maximum count value has been reached:

$$COUNT_{MAX} = 2^{32} - 1$$

for 32-bit counter mode, and

$$COUNT_{MAX} = 2^{16} - 1$$

for 16-bit counter mode.

When running in 16-bit mode, the overflow value is selectable with a period value. The counter overflow will then occur when the counter value reaches the specified period value.

14.3.1. Periodic Events

The RTC can generate events at periodic intervals, allowing for direct peripheral actions without CPU intervention. The periodic events can be generated on the upper eight bits of the RTC prescaler, and will be generated on the rising edge transition of the specified bit. The resulting periodic frequency can be calculated by the following formula:

$$f_{PERIODIC} = \frac{f_{ASY}}{2^{n+3}}$$

Where

f_{ASY}

refers to the *asynchronous* clock is set up in the RTC module configuration. The **n** parameter is the event source generator index of the RTC module. If the asynchronous clock is operated at the recommended frequency of 1KHz, the formula results in the values shown in [Table 14-1](#).

Table 14-1. RTC Event Frequencies for Each Prescaler Bit Using a 1KHz Clock

n	Periodic event
7	1Hz
6	2Hz
5	4Hz
4	8Hz
3	16Hz
2	32Hz
1	64Hz
0	128Hz

Note: The connection of events between modules requires the use of the SAM Event System (EVENTS) Driver to route output event of one module to the the input event of another. For more information on event routing, refer to the event driver documentation.

14.3.2. Digital Frequency Correction

The RTC module contains Digital Frequency Correction logic to compensate for inaccurate source clock frequencies which would otherwise result in skewed time measurements. The correction scheme requires that at least two bits in the RTC module prescaler are reserved by the correction logic. As a result of this implementation, frequency correction is only available when the RTC is running from a 1Hz reference clock.

The correction procedure is implemented by subtracting or adding a single cycle from the RTC prescaler every 1024 RTC GCLK cycles. The adjustment is applied the specified number of time (maximum 127) over 976 of these periods. The corresponding correction in PPM will be given by:

$$Correction(PPM) = \frac{VALUE}{999424} 10^6$$

The RTC clock will tick faster if provided with a positive correction value, and slower when given a negative correction value.

14.3.3. RTC Tamper Detect

see [RTC Tamper Detect](#)

14.4. Special Considerations

14.4.1. Clock Setup

14.4.1.1. SAM D20/D21/R21/D10/D11/DA1 Clock Setup

The RTC is typically clocked by a specialized GCLK generator that has a smaller prescaler than the others. By default the RTC clock is on, selected to use the internal 32KHz RC-oscillator with a prescaler of 32, giving a resulting clock frequency of 1KHz to the RTC. When the internal RTC prescaler is set to 1024, this yields an end-frequency of 1Hz.

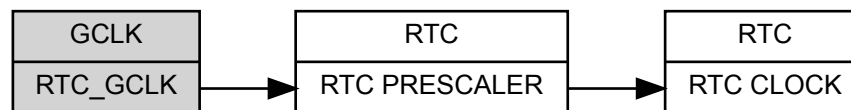
The implementer also has the option to set other end-frequencies. [Table 14-2](#) lists the available RTC frequencies for each possible GCLK and RTC input prescaler options.

Table 14-2. RTC Output Frequencies from Allowable Input Clocks

End-frequency	GCLK prescaler	RTC prescaler
32KHz	1	1
1KHz	32	1
1Hz	32	1024

The overall RTC module clocking scheme is shown in [Figure 14-1](#).

Figure 14-1. SAM D20/D21/R21/D10/D11/DA1 Clock Setup



14.4.1.2. SAM L21/C20/C21 Clock Setup

The RTC clock can be selected from OSC32K, XOSC32K, or OSCULP32K, and a 32KHz or 1KHz oscillator clock frequency is required. This clock must be configured and enabled in the 32KHz oscillator controller before using the RTC.

The table below lists the available RTC clock [Table 14-3](#).

Table 14-3. RTC Clocks Source

RTC clock frequency	Clock source	Description
1.024KHz	ULP1K	1.024KHz from 32KHz internal ULP oscillator
32.768KHz	ULP32K	32.768KHz from 32KHz internal ULP oscillator
1.024KHz	OSC1K	1.024KHz from 32KHz internal oscillator
32.768KHz	OSC32K	32.768KHz from 32KHz internal oscillator
1.024KHz	XOSC1K	1.024KHz from 32KHz internal oscillator
32.768KHz	XOSC32K	32.768KHz from 32KHz external crystal oscillator

14.5. Extra Information

For extra information, see [Extra Information for RTC COUNT Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

14.6. Examples

For a list of examples related to this driver, see [Examples for RTC \(COUNT\) Driver](#).

14.7. API Overview

14.7.1. Structure Definitions

14.7.1.1. Struct `rtc_count_config`

Configuration structure for the RTC instance. This structure should be initialized using the [rtc_count_get_config_defaults\(\)](#) before any user configurations are set.

Table 14-4. Members

Type	Name	Description
bool	clear_on_match	If true, clears the counter value on compare match. Only available whilst running in 32-bit mode
uint32_t	compare_values[]	Array of Compare values. Not all Compare values are available in 32-bit mode
enum rtc_count_mode	mode	Select the operation mode of the RTC
enum rtc_count_prescaler	prescaler	Input clock prescaler for the RTC module

14.7.1.2. Struct `rtc_count_events`

Event flags for the [rtc_count_enable_events\(\)](#) and [rtc_count_disable_events\(\)](#).

Table 14-5. Members

Type	Name	Description
bool	generate_event_on_compare[]	Generate an output event on a compare channel match against the RTC count
bool	generate_event_on_overflow	Generate an output event on each overflow of the RTC count
bool	generate_event_on_periodic[]	Generate an output event periodically at a binary division of the RTC counter frequency

Type	Name	Description
bool	generate_event_on_tamper	Generate an output event on every tamper input
bool	on_event_to_tamper	Tamper input event and capture the COUNT value

14.7.1.3. Struct rtc_tamper_config

The configuration structure for the RTC tamper. This structure should be initialized using the [rtc_tamper_get_config_defaults\(\)](#) before any user configurations are set.

Table 14-6. Members

Type	Name	Description
enum rtc_tamper_active_layer_freq_divider	actl_freq_div	Active layer frequency
bool	bkup_reset_on_tamper	Backup register reset on tamper enable
enum rtc_tamper_debounce_freq_divider	deb_freq_div	Debounce frequency
enum rtc_tamper_debounce_seq	deb_seq	Debounce sequential
bool	dma_tamper_enable	DMA on tamper enable
bool	gp0_enable	General Purpose 0/1 Enable
bool	gp_reset_on_tamper	GP register reset on tamper enable
struct rtc_tamper_input_config	in_cfg[]	Tamper IN configuration

14.7.1.4. Struct rtc_tamper_input_config

The configuration structure for tamper INn.

Table 14-7. Members

Type	Name	Description
enum rtc_tamper_input_action	action	Tamper input action
bool	debounce_enable	Debounce enable
enum rtc_tamper_level_sel	level	Tamper level select

14.7.2. Macro Definitions

14.7.2.1. Driver Feature Definition

Define port features set according to different device family.

Macro FEATURE_RTC_PERIODIC_INT

```
#define FEATURE_RTC_PERIODIC_INT
```

RTC periodic interval interrupt.

Macro FEATURE_RTC_PRESCALER_OFF

```
#define FEATURE_RTC_PRESCALER_OFF
```

RTC prescaler is off.

Macro FEATURE_RTC_CLOCK_SELECTION

```
#define FEATURE_RTC_CLOCK_SELECTION
```

RTC clock selection.

Macro FEATURE_RTC_GENERAL_PURPOSE_REG

```
#define FEATURE_RTC_GENERAL_PURPOSE_REG
```

General purpose registers.

Macro FEATURE_RTC_TAMPER_DETECTION

```
#define FEATURE_RTC_TAMPER_DETECTION
```

RTC tamper detection.

14.7.2.2. Macro RTC_TAMPER_DETECT_EVT

```
#define RTC_TAMPER_DETECT_EVT
```

RTC tamper input event detection bitmask.

14.7.2.3. Macro RTC_TAMPER_DETECT_ID0

```
#define RTC_TAMPER_DETECT_ID0
```

RTC tamper ID0 detection bitmask.

14.7.2.4. Macro RTC_TAMPER_DETECT_ID1

```
#define RTC_TAMPER_DETECT_ID1
```

RTC tamper ID1 detection bitmask.

14.7.2.5. Macro RTC_TAMPER_DETECT_ID2

```
#define RTC_TAMPER_DETECT_ID2
```

RTC tamper ID2 detection bitmask.

14.7.2.6. Macro RTC_TAMPER_DETECT_ID3

```
#define RTC_TAMPER_DETECT_ID3
```

RTC tamper ID3 detection bitmask.

14.7.2.7. Macro RTC_TAMPER_DETECT_ID4

```
#define RTC_TAMPER_DETECT_ID4
```

RTC tamper ID4 detection bitmask.

14.7.3. Function Definitions

14.7.3.1. Configuration and Initialization

Function `rtc_count_get_config_defaults()`

Gets the RTC default configurations.

```
void rtc_count_get_config_defaults(  
    struct rtc_count_config *const config)
```

Initializes the configuration structure to default values. This function should be called at the start of any RTC initialization.

The default configuration is:

- Input clock divided by a factor of 1024
- RTC in 32-bit mode
- Clear on compare match off
- Continuously sync count register off
- No event source on
- All compare values equal 0
- Count read synchronization is enabled for SAM L22

Table 14-8. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to be initialized to default values

Function `rtc_count_reset()`

Resets the RTC module. Resets the RTC to hardware defaults.

```
void rtc_count_reset(  
    struct rtc_module *const module)
```

Table 14-9. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `rtc_count_enable()`

Enables the RTC module.

```
void rtc_count_enable(  
    struct rtc_module *const module)
```

Enables the RTC module once it has been configured, ready for use. Most module configuration parameters cannot be altered while the module is enabled.

Table 14-10. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

Function `rtc_count_disable()`

Disables the RTC module.

```
void rtc_count_disable(  
    struct rtc_module *const module)
```

Disables the RTC module.

Table 14-11. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

Function `rtc_count_init()`

Initializes the RTC module with given configurations.

```
enum status_code rtc_count_init(  
    struct rtc_module *const module,  
    Rtc *const hw,  
    const struct rtc_count_config *const config)
```

Initializes the module, setting up all given configurations to provide the desired functionality of the RTC.

Table 14-12. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Pointer to hardware instance
[in]	config	Pointer to the configuration structure

Returns

Status of the initialization procedure.

Table 14-13. Return Values

Return value	Description
STATUS_OK	If the initialization was run stressfully
STATUS_ERR_INVALID_ARG	If invalid argument(s) were given

Function `rtc_count_frequency_correction()`

Calibrate for too-slow or too-fast oscillator.

```
enum status_code rtc_count_frequency_correction(  
    struct rtc_module *const module,  
    const int8_t value)
```

When used, the RTC will compensate for an inaccurate oscillator. The RTC module will add or subtract cycles from the RTC prescaler to adjust the frequency in approximately 1 PPM steps. The provided correction value should be between 0 and 127, allowing for a maximum 127 PPM correction.

If no correction is needed, set value to zero.

Note: Can only be used when the RTC is operated in 1Hz.

Table 14-14. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	value	Ranging from -127 to 127 used for the correction

Returns

Status of the calibration procedure.

Table 14-15. Return Values

Return value	Description
STATUS_OK	If calibration was executed correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

14.7.3.2. Count and Compare Value Management

Function `rtc_count_set_count()`

Set the current count value to desired value.

```
enum status_code rtc_count_set_count(  
    struct rtc_module *const module,  
    const uint32_t count_value)
```

Sets the value of the counter to the specified value.

Table 14-16. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	count_value	The value to be set in count register

Returns

Status of setting the register.

Table 14-17. Return Values

Return value	Description
STATUS_OK	If everything was executed correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

Function `rtc_count_get_count()`

Get the current count value.

```
uint32_t rtc_count_get_count(  
    struct rtc_module *const module)
```


Table 14-18. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Returns the current count value.

Returns

The current counter value as a 32-bit unsigned integer.

Function rtc_count_set_compare()

Set the compare value for the specified compare.

```
enum status_code rtc_count_set_compare(
    struct rtc_module *const module,
    const uint32_t comp_value,
    const enum rtc_count_compare comp_index)
```

Sets the value specified by the implementer to the requested compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 14-19. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	comp_value	The value to be written to the compare
[in]	comp_index	Index of the compare to set

Returns

Status indicating if compare was successfully set.

Table 14-20. Return Values

Return value	Description
STATUS_OK	If compare was successfully set
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_BAD_FORMAT	If the module was not initialized in a mode

Function rtc_count_get_compare()

Get the current compare value of specified compare.

```
enum status_code rtc_count_get_compare(
    struct rtc_module *const module,
    uint32_t *const comp_value,
    const enum rtc_count_compare comp_index)
```

Retrieves the current value of the specified compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 14-21. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	comp_value	Pointer to 32-bit integer that will be populated with the current compare value
[in]	comp_index	Index of compare to check

Returns

Status of the reading procedure.

Table 14-22. Return Values

Return value	Description
STATUS_OK	If the value was read correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_BAD_FORMAT	If the module was not initialized in a mode

Function rtc_count_set_period()

Set the given value to the period.

```
enum status_code rtc_count_set_period(
    struct rtc_module *const module,
    uint16_t period_value)
```

Sets the given value to the period.

Note: Only available in 16-bit mode.

Table 14-23. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	period_value	The value to set to the period

Returns

Status of setting the period value.

Table 14-24. Return Values

Return value	Description
STATUS_OK	If the period was set correctly
STATUS_ERR_UNSUPPORTED_DEV	If module is not operated in 16-bit mode

Function `rtc_count_get_period()`

Retrieves the value of period.

```
enum status_code rtc_count_get_period(  
    struct rtc_module *const module,  
    uint16_t *const period_value)
```

Retrieves the value of the period for the 16-bit mode counter.

Note: Only available in 16-bit mode.

Table 14-25. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	period_value	Pointer to value for return argument

Returns

Status of getting the period value.

Table 14-26. Return Values

Return value	Description
STATUS_OK	If the period value was read correctly
STATUS_ERR_UNSUPPORTED_DEV	If incorrect mode was set

14.7.3.3. Status Management

Function `rtc_count_is_overflow()`

Check if an RTC overflow has occurred.

```
bool rtc_count_is_overflow(  
    struct rtc_module *const module)
```

Checks the overflow flag in the RTC. The flag is set when there is an overflow in the clock.

Table 14-27. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

Returns

Overflow state of the RTC module.

Table 14-28. Return Values

Return value	Description
true	If the RTC count value has overflowed
false	If the RTC count value has not overflowed

Function `rtc_count_clear_overflow()`

Clears the RTC overflow flag.

```
void rtc_count_clear_overflow(  
    struct rtc_module *const module)
```

Clears the RTC module counter overflow flag, so that new overflow conditions can be detected.

Table 14-29. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

Function `rtc_count_is_periodic_interval()`

Check if an RTC periodic interval interrupt has occurred.

```
bool rtc_count_is_periodic_interval(  
    struct rtc_module *const module,  
    enum rtc_count_periodic_interval n)
```

Checks the periodic interval flag in the RTC.

Table 14-30. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	n	RTC periodic interval interrupt

Returns

Periodic interval interrupt state of the RTC module.

Table 14-31. Return Values

Return value	Description
true	RTC periodic interval interrupt occurs
false	RTC periodic interval interrupt doesn't occur

Function `rtc_count_clear_periodic_interval()`

Clears the RTC periodic interval flag.

```
void rtc_count_clear_periodic_interval(  
    struct rtc_module *const module,  
    enum rtc_count_periodic_interval n)
```

Clears the RTC module counter periodic interval flag, so that new periodic interval conditions can be detected.

Table 14-32. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	n	RTC periodic interval interrupt

Function `rtc_count_is_compare_match()`

Check if RTC compare match has occurred.

```
bool rtc_count_is_compare_match(
    struct rtc_module *const module,
    const enum rtc_count_compare comp_index)
```

Checks the compare flag to see if a match has occurred. The compare flag is set when there is a compare match between counter and the compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 14-33. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	comp_index	Index of compare to check current flag

Function `rtc_count_clear_compare_match()`

Clears RTC compare match flag.

```
enum status_code rtc_count_clear_compare_match(
    struct rtc_module *const module,
    const enum rtc_count_compare comp_index)
```

Clears the compare flag. The compare flag is set when there is a compare match between the counter and the compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 14-34. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	comp_index	Index of compare to check current flag

Returns

Status indicating if flag was successfully cleared.

Table 14-35. Return Values

Return value	Description
STATUS_OK	If flag was successfully cleared
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_BAD_FORMAT	If the module was not initialized in a mode

14.7.3.4. Event Management

Function `rtc_count_enable_events()`

Enables an RTC event output.

```
void rtc_count_enable_events(
    struct rtc_module *const module,
    struct rtc_count_events *const events)
```

Enables one or more output events from the RTC module. See [rtc_count_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 14-36. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	events	Struct containing flags of events to enable

Function `rtc_count_disable_events()`

Disables an RTC event output.

```
void rtc_count_disable_events(
    struct rtc_module *const module,
    struct rtc_count_events *const events)
```

Disabled one or more output events from the RTC module. See [rtc_count_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 14-37. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	events	Struct containing flags of events to disable

14.7.3.5. RTC General Purpose Registers

Function `rtc_write_general_purpose_reg()`

Write a value into general purpose register.

```
void rtc_write_general_purpose_reg(
    struct rtc_module *const module,
```

```
const uint8_t index,
uint32_t value)
```

Table 14-38. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	n	General purpose type
[in]	index	General purpose register index (0..3)

Function `rtc_read_general_purpose_reg()`

Read the value from general purpose register.

```
uint32_t rtc_read_general_purpose_reg(
    struct rtc_module *const module,
    const uint8_t index)
```

Table 14-39. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	index	General purpose register index (0..3)

Returns

Value of general purpose register.

14.7.3.6. Callbacks

Function `rtc_count_register_callback()`

Registers callback for the specified callback type.

```
enum status_code rtc_count_register_callback(
    struct rtc_module *const module,
    rtc_count_callback_t callback,
    enum rtc_count_callback callback_type)
```

Associates the given callback function with the specified callback type. To enable the callback, the [rtc_count_enable_callback](#) function must be used.

Table 14-40. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback	Pointer to the function desired for the specified callback
[in]	callback_type	Callback type to register

Returns

Status of registering callback.

Table 14-41. Return Values

Return value	Description
STATUS_OK	Registering was done successfully
STATUS_ERR_INVALID_ARG	If trying to register a callback not available

Function `rtc_count_unregister_callback()`

Unregisters callback for the specified callback type.

```
enum status_code rtc_count_unregister_callback(
    struct rtc_module *const module,
    enum rtc_count_callback callback_type)
```

When called, the currently registered callback for the given callback type will be removed.

Table 14-42. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Specifies the callback type to unregister

Returns

Status of unregistering callback.

Table 14-43. Return Values

Return value	Description
STATUS_OK	Unregistering was done successfully
STATUS_ERR_INVALID_ARG	If trying to unregister a callback not available

Function `rtc_count_enable_callback()`

Enables callback.

```
void rtc_count_enable_callback(
    struct rtc_module *const module,
    enum rtc_count_callback callback_type)
```

Enables the callback specified by the `callback_type`.

Table 14-44. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Callback type to enable

Function `rtc_count_disable_callback()`

Disables callback.

```
void rtc_count_disable_callback(  
    struct rtc_module *const module,  
    enum rtc_count_callback callback_type)
```

Disables the callback specified by the `callback_type`.

Table 14-45. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Callback type to disable

14.7.3.7. RTC Tamper Detection

Function `rtc_tamper_get_config_defaults()`

Gets the RTC tamper default configurations.

```
void rtc_tamper_get_config_defaults(  
    struct rtc_tamper_config *const config)
```

Initializes the configuration structure to default values.

The default configuration is as follows:

- Disable backup register reset on tamper
- Disable GP register reset on tamper
- Active layer clock divided by a factor of 8
- Debounce clock divided by a factor of 8
- Detect edge on INn with synchronous stability debouncing
- Disable DMA on tamper
- Enable GP register
- Disable debounce, detect on falling edge and no action on INn

Table 14-46. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to be initialized to default values.

Function `rtc_tamper_set_config()`

```
enum status_code rtc_tamper_set_config(  
    struct rtc_module *const module,  
    struct rtc_tamper_config *const tamper_cfg)
```

Function `rtc_tamper_get_detect_flag()`

Retrieves the RTC tamper detection status.

```
uint32_t rtc_tamper_get_detect_flag(  
    struct rtc_module *const module)
```

Retrieves the detection status of each input pin and the input event.

Table 14-47. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the RTC software instance struct

Returns

Bitmask of detection flags.

Table 14-48. Return Values

Return value	Description
RTC_TAMPER_DETECT_ID0	Tamper condition on IN0 has been detected
RTC_TAMPER_DETECT_ID1	Tamper condition on IN1 has been detected
RTC_TAMPER_DETECT_ID2	Tamper condition on IN2 has been detected
RTC_TAMPER_DETECT_ID3	Tamper condition on IN3 has been detected
RTC_TAMPER_DETECT_ID4	Tamper condition on IN4 has been detected
RTC_TAMPER_DETECT_EVT	Tamper input event has been detected

Function rtc_tamper_clear_detect_flag()

Clears RTC tamper detection flag.

```
void rtc_tamper_clear_detect_flag(
    struct rtc_module *const module,
    const uint32_t detect_flags)
```

Clears the given detection flag of the module.

Table 14-49. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the TC software instance struct
[in]	detect_flags	Bitmask of detection flags

14.7.3.8. Function rtc_tamper_get_stamp()

Get the tamper stamp value.

```
uint32_t rtc_tamper_get_stamp(
    struct rtc_module *const module)
```

Table 14-50. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Returns

The current tamper stamp value as a 32-bit unsigned integer.

14.7.4. Enumeration Definitions

14.7.4.1. Enum rtc_clock_sel

Table 14-51. Members

Enum value	Description
RTC_CLOCK_SELECTION_ULP1K	1.024KHz from 32KHz internal ULP oscillator
RTC_CLOCK_SELECTION_ULP32K	32.768KHz from 32KHz internal ULP oscillator
RTC_CLOCK_SELECTION_OSC1K	1.024KHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_OSC32K	32.768KHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_XOSC1K	1.024KHz from 32KHz external oscillator
RTC_CLOCK_SELECTION_XOSC32K	32.768KHz from 32.768KHz external crystal oscillator

14.7.4.2. Enum rtc_count_callback

The available callback types for the RTC count module.

Table 14-52. Members

Enum value	Description
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_0	Callback for Periodic Interval 0 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_1	Callback for Periodic Interval 1 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_2	Callback for Periodic Interval 2 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_3	Callback for Periodic Interval 3 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_4	Callback for Periodic Interval 4 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_5	Callback for Periodic Interval 5 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_6	Callback for Periodic Interval 6 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_7	Callback for Periodic Interval 7 Interrupt
RTC_COUNT_CALLBACK_COMPARE_0	Callback for compare channel 0
RTC_COUNT_CALLBACK_COMPARE_1	Callback for compare channel 1
RTC_COUNT_CALLBACK_COMPARE_2	Callback for compare channel 2
RTC_COUNT_CALLBACK_COMPARE_3	Callback for compare channel 3
RTC_COUNT_CALLBACK_COMPARE_4	Callback for compare channel 4
RTC_COUNT_CALLBACK_COMPARE_5	Callback for compare channel 5
RTC_COUNT_CALLBACK_TAMPER	Callback for tamper
RTC_COUNT_CALLBACK_OVERFLOW	Callback for overflow

14.7.4.3. Enum rtc_count_compare

Note: Not all compare channels are available in all devices and modes.

Table 14-53. Members

Enum value	Description
RTC_COUNT_COMPARE_0	Compare channel 0
RTC_COUNT_COMPARE_1	Compare channel 1
RTC_COUNT_COMPARE_2	Compare channel 2
RTC_COUNT_COMPARE_3	Compare channel 3
RTC_COUNT_COMPARE_4	Compare channel 4
RTC_COUNT_COMPARE_5	Compare channel 5

14.7.4.4. Enum rtc_count_mode

RTC Count operating modes, to select the counting width and associated module operation.

Table 14-54. Members

Enum value	Description
RTC_COUNT_MODE_16BIT	RTC Count module operates in 16-bit mode
RTC_COUNT_MODE_32BIT	RTC Count module operates in 32-bit mode

14.7.4.5. Enum rtc_count_periodic_interval

Table 14-55. Members

Enum value	Description
RTC_COUNT_PERIODIC_INTERVAL_0	Periodic interval 0
RTC_COUNT_PERIODIC_INTERVAL_1	Periodic interval 1
RTC_COUNT_PERIODIC_INTERVAL_2	Periodic interval 2
RTC_COUNT_PERIODIC_INTERVAL_3	Periodic interval 3
RTC_COUNT_PERIODIC_INTERVAL_4	Periodic interval 4
RTC_COUNT_PERIODIC_INTERVAL_5	Periodic interval 5
RTC_COUNT_PERIODIC_INTERVAL_6	Periodic interval 6
RTC_COUNT_PERIODIC_INTERVAL_7	Periodic interval 7

14.7.4.6. Enum rtc_count_prescaler

The available input clock prescaler values for the RTC count module.

Table 14-56. Members

Enum value	Description
RTC_COUNT_PRESCALER_OFF	RTC prescaler is off, and the input clock frequency is prescaled by a factor of 1
RTC_COUNT_PRESCALER_DIV_1	RTC input clock frequency is prescaled by a factor of 1
RTC_COUNT_PRESCALER_DIV_2	RTC input clock frequency is prescaled by a factor of 2
RTC_COUNT_PRESCALER_DIV_4	RTC input clock frequency is prescaled by a factor of 4
RTC_COUNT_PRESCALER_DIV_8	RTC input clock frequency is prescaled by a factor of 8
RTC_COUNT_PRESCALER_DIV_16	RTC input clock frequency is prescaled by a factor of 16
RTC_COUNT_PRESCALER_DIV_32	RTC input clock frequency is prescaled by a factor of 32
RTC_COUNT_PRESCALER_DIV_64	RTC input clock frequency is prescaled by a factor of 64
RTC_COUNT_PRESCALER_DIV_128	RTC input clock frequency is prescaled by a factor of 128
RTC_COUNT_PRESCALER_DIV_256	RTC input clock frequency is prescaled by a factor of 256
RTC_COUNT_PRESCALER_DIV_512	RTC input clock frequency is prescaled by a factor of 512
RTC_COUNT_PRESCALER_DIV_1024	RTC input clock frequency is prescaled by a factor of 1024

14.7.4.7. Enum rtc_tamper_active_layer_freq_divider

The available prescaler factor for the RTC clock output used during active layer protection.

Table 14-57. Members

Enum value	Description
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_2	RTC active layer frequency is prescaled by a factor of 2
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_4	RTC active layer frequency is prescaled by a factor of 4
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_8	RTC active layer frequency is prescaled by a factor of 8
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_16	RTC active layer frequency is prescaled by a factor of 16
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_32	RTC active layer frequency is prescaled by a factor of 32
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_64	RTC active layer frequency is prescaled by a factor of 64

Enum value	Description
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_128	RTC active layer frequency is prescaled by a factor of 128
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_256	RTC active layer frequency is prescaled by a factor of 256

14.7.4.8. Enum `rtc_tamper_debounce_freq_divider`

The available prescaler factor for the input debouncers.

Table 14-58. Members

Enum value	Description
RTC_TAMPER_DEBOUNCE_FREQ_DIV_2	RTC debounce frequency is prescaled by a factor of 2
RTC_TAMPER_DEBOUNCE_FREQ_DIV_4	RTC debounce frequency is prescaled by a factor of 4
RTC_TAMPER_DEBOUNCE_FREQ_DIV_8	RTC debounce frequency is prescaled by a factor of 8
RTC_TAMPER_DEBOUNCE_FREQ_DIV_16	RTC debounce frequency is prescaled by a factor of 16
RTC_TAMPER_DEBOUNCE_FREQ_DIV_32	RTC debounce frequency is prescaled by a factor of 32
RTC_TAMPER_DEBOUNCE_FREQ_DIV_64	RTC debounce frequency is prescaled by a factor of 64
RTC_TAMPER_DEBOUNCE_FREQ_DIV_128	RTC debounce frequency is prescaled by a factor of 128
RTC_TAMPER_DEBOUNCE_FREQ_DIV_256	RTC debounce frequency is prescaled by a factor of 256

14.7.4.9. Enum `rtc_tamper_debounce_seq`

The available sequential for tamper debounce.

Table 14-59. Members

Enum value	Description
RTC_TAMPER_DEBOUNCE_SYNC	Tamper input detect edge with synchronous stability debounce
RTC_TAMPER_DEBOUNCE_ASYNC	Tamper input detect edge with asynchronous stability debounce
RTC_TAMPER_DEBOUNCE_MAJORITY	Tamper input detect edge with majority debounce

14.7.4.10. Enum `rtc_tamper_input_action`

The available action taken by the tamper input.

Table 14-60. Members

Enum value	Description
RTC_TAMPER_INPUT_ACTION_OFF	RTC tamper input action is disabled
RTC_TAMPER_INPUT_ACTION_WAKE	RTC tamper input action is wake and set tamper flag
RTC_TAMPER_INPUT_ACTION_CAPTURE	RTC tamper input action is capture timestamp and set tamper flag
RTC_TAMPER_INPUT_ACTION_ACTL	RTC tamper input action is compare IN to OUT, when a mismatch occurs, capture timestamp and set tamper flag

14.7.4.11. Enum rtc_tamper_level_sel

The available edge condition for tamper INn level select.

Table 14-61. Members

Enum value	Description
RTC_TAMPER_LEVEL_FALLING	A falling edge condition will be detected on Tamper input
RTC_TAMPER_LEVEL_RISING	A rising edge condition will be detected on Tamper input

14.8. RTC Tamper Detect

The RTC provides several selectable polarity external inputs (INn) that can be used for tamper detection. When detect, tamper inputs support the four actions:

- Off
- Wake
- Capture
- Active layer protection

Note: The Active Layer Protection is a means of detecting broken traces on the PCB provided by RTC. In this mode an RTC output signal is routed over critical components on the board and fed back to one of the RTC inputs. The input and output signals are compared and a tamper condition is detected when they do not match.

Separate debouncers are embedded for each external input. The detection time depends on whether the debouncer operates synchronously or asynchronously, and whether majority detection is enabled or not. For details, refer to the section "Tamper Detection" of datasheet.

14.9. Extra Information for RTC COUNT Driver

14.9.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
RTC	Real Time Counter
PPM	Part Per Million
RC	Resistor/Capacitor

14.9.2. Dependencies

This driver has the following dependencies:

- None

14.9.3. Errata

There are no errata related to this driver.

14.9.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added support for SAM C21
Added support for SAM L21/L22
Added support for RTC tamper feature
Added driver instance parameter to all API function calls, except <code>get_config_defaults</code>
Updated initialization function to also enable the digital interface clock to the module if it is disabled
Initial Release

14.10. Examples for RTC (COUNT) Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM RTC Count \(RTC COUNT\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for RTC \(COUNT\) - Basic](#)
- [Quick Start Guide for RTC \(COUNT\) - Callback](#)
- [Quick Start Guide for RTC Tamper with DMA](#)

14.10.1. Quick Start Guide for RTC (COUNT) - Basic

In this use case, the RTC is set up in count mode. The example configures the RTC in 16-bit mode, with continuous updates to the COUNT register, together with a set compare register value. Every 2000ms a LED on the board is toggled.

14.10.1.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

Clocks and Oscillators

The `conf_clock.h` file needs to be changed with the following values to configure the clocks and oscillators for the module.

The following oscillator settings are needed:

```
/* SYSTEM_CLOCK_SOURCE_OSC32K configuration - Internal 32KHz oscillator */
# define CONF_CLOCK_OSC32K_ENABLE           true
# define CONF_CLOCK_OSC32K_STARTUP_TIME     SYSTEM_OSC32K_STARTUP_130
# define CONF_CLOCK_OSC32K_ENABLE_1KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ENABLE_32KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ON_DEMAND        true
# define CONF_CLOCK_OSC32K_RUN_IN_STANDBY   false
```

The following generic clock settings are needed:

```
/* Configure GCLK generator 2 (RTC) */
# define CONF_CLOCK_GCLK_2_ENABLE           true
# define CONF_CLOCK_GCLK_2_RUN_IN_STANDBY   false
# define CONF_CLOCK_GCLK_2_CLOCK_SOURCE     SYSTEM_CLOCK_SOURCE_OSC32K
# define CONF_CLOCK_GCLK_2_PRESCALER        32
# define CONF_CLOCK_GCLK_2_OUTPUT_ENABLE    false
```

14.10.1.2. Setup

Initialization Code

Create an `rtc_module` struct and add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

Copy-paste the following setup code to your applications `main()`:

```
void configure_rtc_count(void)
{
    struct rtc_count_config config_rtc_count;

    rtc_count_get_config_defaults(&config_rtc_count);

    config_rtc_count.prescaler      = RTC_COUNT_PRESCALER_DIV_1;
    config_rtc_count.mode           = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif
    config_rtc_count.compare_values[0] = 1000;
    rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

    rtc_count_enable(&rtc_instance);
}
```

Add to Main

Add the following to your `main()`.

```
configure_rtc_count();
```

Workflow

1. Create an RTC configuration structure to hold the desired RTC driver settings.

```
struct rtc_count_config config_rtc_count;
```

2. Fill the configuration structure with the default driver configuration.

```
rtc_count_get_config_defaults(&config_rtc_count);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the RTC driver configuration to run in 16-bit counting mode, with continuous counter register updates.

```
config_rtc_count.prescaler          = RTC_COUNT_PRESCALER_DIV_1;
config_rtc_count.mode               = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif
config_rtc_count.compare_values[0]  = 1000;
```

4. Initialize the RTC module.

```
rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
```

5. Enable the RTC module, so that it may begin counting.

```
rtc_count_enable(&rtc_instance);
```

14.10.1.3. Implementation

Code used to implement the initialized module.

Code

Add after initialization in main().

```
rtc_count_set_period(&rtc_instance, 2000);

while (true) {
    if (rtc_count_is_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0)) {
        /* Do something on RTC count match here */
        port_pin_toggle_output_level(LED_0_PIN);

        rtc_count_clear_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0);
    }
}
```

Workflow

1. Set RTC period to 2000ms (two seconds) so that it will overflow and reset back to zero every two seconds.

```
rtc_count_set_period(&rtc_instance, 2000);
```

2. Enter an infinite loop to poll the RTC driver to check when a comparison match occurs.

```
while (true) {
```

3. Check if the RTC driver has found a match on compare channel 0 against the current RTC count value.

```
if (rtc_count_is_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0)) {
```

4. Once a compare match occurs, perform the desired user action.

```
/* Do something on RTC count match here */
port_pin_toggle_output_level(LED_0_PIN);
```

5. Clear the compare match, so that future matches may occur.

```
rtc_count_clear_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0);
```

14.10.2. Quick Start Guide for RTC (COUNT) - Callback

In this use case, the RTC is set up in count mode. The quick start configures the RTC in 16-bit mode and to continuously update COUNT register. The rest of the configuration is according to the [default](#). A callback is implemented for when the RTC overflows.

14.10.2.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

Clocks and Oscillators

The `conf_clock.h` file needs to be changed with the following values to configure the clocks and oscillators for the module.

The following oscillator settings are needed:

```
/* SYSTEM_CLOCK_SOURCE_OSC32K configuration - Internal 32KHz oscillator */
# define CONF_CLOCK_OSC32K_ENABLE true
# define CONF_CLOCK_OSC32K_STARTUP_TIME SYSTEM_OSC32K_STARTUP_130
# define CONF_CLOCK_OSC32K_ENABLE_1KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ENABLE_32KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ON_DEMAND true
# define CONF_CLOCK_OSC32K_RUN_IN_STANDBY false
```

The following generic clock settings are needed:

```
/* Configure GCLK generator 2 (RTC) */
# define CONF_CLOCK_GCLK_2_ENABLE true
# define CONF_CLOCK_GCLK_2_RUN_IN_STANDBY false
# define CONF_CLOCK_GCLK_2_CLOCK_SOURCE SYSTEM_CLOCK_SOURCE_OSC32K
# define CONF_CLOCK_GCLK_2_PRESCALER 32
# define CONF_CLOCK_GCLK_2_OUTPUT_ENABLE false
```

14.10.2.2. Setup

Code

Create an `rtc_module` struct and add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

The following must be added to the user application:

Function for setting up the module:

```
void configure_rtc_count(void)
{
    struct rtc_count_config config_rtc_count;
    rtc_count_get_config_defaults(&config_rtc_count);

    config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
    config_rtc_count.mode = RTC_COUNT_MODE_16BIT;
```

```
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif
rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

rtc_count_enable(&rtc_instance);
}
```

Callback function:

```
void rtc_overflow_callback(void)
{
    /* Do something on RTC overflow here */
    port_pin_toggle_output_level(LED_0_PIN);
}
```

Function for setting up the callback functionality of the driver:

```
void configure_rtc_callbacks(void)
{
    rtc_count_register_callback(
        &rtc_instance, rtc_overflow_callback,
        RTC_COUNT_CALLBACK_OVERFLOW);
    rtc_count_enable_callback(&rtc_instance, RTC_COUNT_CALLBACK_OVERFLOW);
}
```

Add to user application main():

```
/* Initialize system. Must configure conf_clocks.h first. */
system_init();

/* Configure and enable RTC */
configure_rtc_count();

/* Configure and enable callback */
configure_rtc_callbacks();

/* Set period */
rtc_count_set_period(&rtc_instance, 2000);
```

Workflow

1. Initialize system.

```
system_init();
```

2. Configure and enable module.

```
configure_rtc_count();
```

3. Create an RTC configuration structure to hold the desired RTC driver settings and fill it with the default driver configuration values.

```
struct rtc_count_config config_rtc_count;
rtc_count_get_config_defaults(&config_rtc_count);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Alter the RTC driver configuration to run in 16-bit counting mode, with continuous counter register updates and a compare value of 1000ms.

```
config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
config_rtc_count.mode = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
```

```
config_rtc_count.continuously_update = true;
#endif
```

5. Initialize the RTC module.

```
rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
```

6. Enable the RTC module, so that it may begin counting.

```
rtc_count_enable(&rtc_instance);
```

7. Configure callback functionality.

```
configure_rtc_callbacks();
```

1. Register overflow callback.

```
rtc_count_register_callback(
    &rtc_instance, rtc_overflow_callback,
    RTC_COUNT_CALLBACK_OVERFLOW);
```

2. Enable overflow callback.

```
rtc_count_enable_callback(&rtc_instance,
    RTC_COUNT_CALLBACK_OVERFLOW);
```

8. Set period.

```
rtc_count_set_period(&rtc_instance, 2000);
```

14.10.2.3. Implementation

Code

Add to user application main:

```
while (true) {
    /* Infinite while loop */
}
```

Workflow

1. Infinite while loop while waiting for callbacks.

```
while (true) {
    /* Infinite while loop */
}
```

14.10.2.4. Callback

Each time the RTC counter overflows, the callback function will be called.

Workflow

1. Perform the desired user action for each RTC overflow:

```
/* Do something on RTC overflow here */
port_pin_toggle_output_level(LED_0_PIN);
```

14.10.3. Quick Start Guide for RTC Tamper with DMA

In this use case, the RTC is set up in count mode. The quick start configures the RTC in 32-bit mode and . The rest of the configuration is according to the [default](#). A callback is implemented for when the RTC capture tamper stamp.

14.10.3.1. Setup

Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

Code

Add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

```
struct dma_resource example_resource;
```

```
COMPILER_ALIGNED(16)  
DmacDescriptor example_descriptor SECTION_DMAC_DESCRIPTOR;
```

The following must be added to the user application: Function for setting up the module:

```
void configure_rtc(void)  
{  
    struct rtc_count_config config_rtc_count;  
    rtc_count_get_config_defaults(&config_rtc_count);  
    config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;  
    rtc_count_init(&rtc_instance, RTC, &config_rtc_count);  
  
    struct rtc_tamper_config config_rtc_tamper;  
    rtc_tamper_get_config_defaults(&config_rtc_tamper);  
    config_rtc_tamper.dma_tamper_enable = true;  
    config_rtc_tamper.in_cfg[0].level = RTC_TAMPER_LEVEL_RISING;  
    config_rtc_tamper.in_cfg[0].action = RTC_TAMPER_INPUT_ACTION_CAPTURE;  
    rtc_tamper_set_config(&rtc_instance, &config_rtc_tamper);  
  
    rtc_count_enable(&rtc_instance);  
}
```

Callback function:

```
void rtc_tamper_callback(void)  
{  
    /* Do something on RTC tamper capture here */  
    LED_On(LED_0_PIN);  
}
```

Function for setting up the callback functionality of the driver:

```
void configure_rtc_callbacks(void)  
{  
    rtc_count_register_callback(  
        &rtc_instance, rtc_tamper_callback, RTC_COUNT_CALLBACK_TAMPER);  
    rtc_count_enable_callback(&rtc_instance, RTC_COUNT_CALLBACK_TAMPER);  
}
```

Add to user application initialization (typically the start of `main()`):

```
/* Initialize system. Must configure conf_clocks.h first. */  
system_init();  
  
/* Configure and enable RTC */  
configure_rtc();  
  
/* Configure and enable callback */
```

```

configure_rtc_callbacks();
configure_dma_resource(&example_resource);
setup_transfer_descriptor(&example_descriptor);
dma_add_descriptor(&example_resource, &example_descriptor);
while (true) {
    /* Infinite while loop */
}

```

Workflow

1. Initialize system.

```
system_init();
```

2. Configure and enable module.

```
configure_rtc();
```

1. Create a RTC configuration structure to hold the desired RTC driver settings and fill it with the configuration values.

```

struct rtc_count_config config_rtc_count;
rtc_count_get_config_defaults(&config_rtc_count);
config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

2. Initialize the RTC module.

```

struct rtc_count_config config_rtc_count;
rtc_count_get_config_defaults(&config_rtc_count);
config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

```

3. Create a RTC tamper configuration structure and fill it with the configuration values.

```

struct rtc_tamper_config config_rtc_tamper;
rtc_tamper_get_config_defaults(&config_rtc_tamper);
config_rtc_tamper.dma_tamper_enable = true;
config_rtc_tamper.in_cfg[0].level = RTC_TAMPER_LEVEL_RISING;
config_rtc_tamper.in_cfg[0].action =
    RTC_TAMPER_INPUT_ACTION_CAPTURE;
rtc_tamper_set_config(&rtc_instance, &config_rtc_tamper);

```

4. Enable the RTC module, so that it may begin counting.

```
rtc_count_enable(&rtc_instance);
```

3. Configure callback functionality.

```
configure_rtc_callbacks();
```

1. Register overflow callback.

```

rtc_count_register_callback(
    &rtc_instance, rtc_tamper_callback,
    RTC_COUNT_CALLBACK_TAMPER);

```

2. Enable overflow callback.

```
rtc_count_enable_callback(&rtc_instance,  
RTC_COUNT_CALLBACK_TAMPER);
```

4. Configure the DMA.

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

2. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra configurations for the DMA resource. ADC_DMAC_ID_RESRDY trigger causes a beat transfer in this example.

```
config.peripheral_trigger = RTC_DMAC_ID_TIMESTAMP;  
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

4. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

5. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

6. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

7. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;  
descriptor_config.dst_increment_enable = false;  
descriptor_config.src_increment_enable = false;  
descriptor_config.block_transfer_count = 1;  
descriptor_config.source_address = (uint32_t)(&rtc_instance.hw->MODE0.TIMESTAMP.reg);  
descriptor_config.destination_address = (uint32_t)(buffer_rtc_tamper);  
descriptor_config.next_descriptor_address = (uint32_t)descriptor;
```

8. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

9. Add DMA descriptor to DMA resource.

```
dma_add_descriptor(&example_resource, &example_descriptor);
```

14.10.3.2. Implementation

Code

Add to user application main:

1. Infinite while loop while waiting for callbacks.

```
while (true) {  
    /* Infinite while loop */  
}
```

Callback

When the RTC tamper captured, the callback function will be called.

1. LED0 on for RTC tamper capture:

```
/* Do something on RTC tamper capture here */  
LED_On(LED_0_PIN);
```

15. SAM RTC Calendar (RTC CAL) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Real Time Clock functionality in Calendar operating mode, for the configuration and retrieval of the current time and date as maintained by the RTC module. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- RTC (Real Time Clock)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

15.1. Prerequisites

There are no prerequisites for this module.

15.2. Module Overview

The RTC module in the SAM devices is a 32-bit counter, with a 10-bit programmable prescaler. Typically, the RTC clock is run continuously, including in the device's low-power sleep modes, to track the current time and date information. The RTC can be used as a source to wake up the system at a scheduled time or periodically using the alarm functions.

In this driver, the RTC is operated in Calendar mode. This allows for an easy integration of a real time clock and calendar into a user application to track the passing of time and/or perform scheduled tasks.

Whilst operating in Calendar mode, the RTC features:

- Time tracking in seconds, minutes, and hours
- 12 or 24 hour mode
- Date tracking in day, month, and year
- Automatic leap year correction

15.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_RTC_PERIODIC_INT	SAM L21/L22/C20/C21
FEATURE_RTC_PRESCALER_OFF	SAM L21/L22/C20/C21
FEATURE_RTC_CLOCK_SELECTION	SAM L21/L22/C20/C21
FEATURE_RTC_GENERAL_PURPOSE_REG	SAM L21/L22
FEATURE_RTC_CONTINUOUSLY_UPDATED	SAM D20, SAM D21, SAM R21, SAM D10, SAM D11, SAM DA1
FEATURE_RTC_TAMPER_DETECTION	SAM L22

Note: The specific features are only available in the driver when the selected device supports those features.

15.2.2. Alarms and Overflow

The RTC has up to four independent hardware alarms that can be configured by the user application. These alarms will be triggered on match with the current clock value, and can be set up to trigger an interrupt, event, or both. The RTC can also be configured to clear the clock value on alarm match, resetting the clock to the original start time.

If the RTC is operated in clock-only mode (i.e. with calendar disabled), the RTC counter value will instead be cleared on overflow once the maximum count value has been reached:

$$COUNT_{MAX} = 2^{32} - 1$$

When the RTC is operated with the calendar enabled and run using a nominal 1Hz input clock frequency, a register overflow will occur after 64 years.

15.2.3. Periodic Events

The RTC can generate events at periodic intervals, allowing for direct peripheral actions without CPU intervention. The periodic events can be generated on the upper eight bits of the RTC prescaler, and will be generated on the rising edge transition of the specified bit. The resulting periodic frequency can be calculated by the following formula:

$$f_{PERIODIC} = \frac{f_{ASY}}{2^{n+3}}$$

Where

f_{ASY}

refers to the *asynchronous* clock set up in the RTC module configuration. For the RTC to operate correctly in calendar mode, this frequency must be 1KHz, while the RTC's internal prescaler should be set to divide by 1024. The **n** parameter is the event source generator index of the RTC module. If the asynchronous clock is operated at the recommended 1KHz, the formula results in the values shown in [Table 15-1](#).

Table 15-1. RTC Event Frequencies for Each Prescaler Bit Using a 1KHz Clock

n	Periodic event
7	1Hz
6	2Hz
5	4Hz
4	8Hz
3	16Hz
2	32Hz
1	64Hz
0	128Hz

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the input event of another. For more information on event routing, refer to the event driver documentation.

15.2.4. Digital Frequency Correction

The RTC module contains Digital Frequency Correction logic to compensate for inaccurate source clock frequencies which would otherwise result in skewed time measurements. The correction scheme requires that at least two bits in the RTC module prescaler are reserved by the correction logic. As a result of this implementation, frequency correction is only available when the RTC is running from a 1Hz reference clock.

The correction procedure is implemented by subtracting or adding a single cycle from the RTC prescaler every 1024 RTC Generic Clock (GCLK) cycles. The adjustment is applied the specified number of time (maximum 127) over 976 of these periods. The corresponding correction in parts per million (PPM) will be given by:

$$Correction(PPM) = \frac{VALUE}{999424} 10^6$$

The RTC clock will tick faster if provided with a positive correction value, and slower when given a negative correction value.

15.2.5. RTC Tamper Detect

See [RTC Tamper Detect](#).

15.3. Special Considerations

15.3.1. Year Limit

The RTC module has a year range of 63 years from the starting year configured when the module is initialized. Dates outside the start to end year range described below will need software adjustment:

$$[YEAR_{START}, YEAR_{START} + 64]$$

15.3.2. Clock Setup

15.3.2.1. SAM D20/D21/R21/D10/D11/DA1 Clock Setup

The RTC is typically clocked by a specialized GCLK generator that has a smaller prescaler than the others. By default the RTC clock is on, selected to use the internal 32KHz Resistor/Capacitor (RC)-oscillator with a prescaler of 32, giving a resulting clock frequency of 1024Hz to the RTC. When the internal RTC prescaler is set to 1024, this yields an end-frequency of 1Hz for correct time keeping operations.

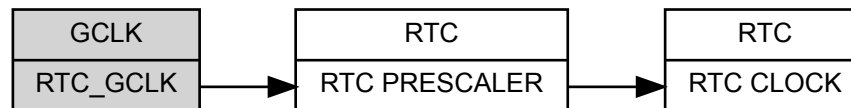
The implementer also has the option to set other end-frequencies. [Table 15-2](#) lists the available RTC frequencies for each possible GCLK and RTC input prescaler options.

Table 15-2. RTC Output Frequencies from Allowable Input Clocks

End-frequency	GCLK prescaler	RTC prescaler
32KHz	1	1
1KHz	32	1
1Hz	32	1024

The overall RTC module clocking scheme is shown in [Figure 15-1](#).

Figure 15-1. SAM D20/D21/R21/D10/D11/DA1 Clock Setup



Note: For the calendar to operate correctly, an asynchronous clock of 1Hz should be used.

15.3.2.2. SAM L21/C20/C21 Clock Setup

The RTC clock can be selected from OSC32K, XOSC32K, or OSCULP32K. A 32KHz or 1KHz oscillator clock frequency is required. This clock must be configured and enabled in the 32KHz oscillator controller before using the RTC.

[Table 15-3](#) lists the available RTC clock.

Table 15-3. RTC Clocks Source

RTC clock frequency	Clock source	Description
1.024kHz	ULP1K	1.024kHz from 32KHz internal ULP oscillator
32.768kHz	ULP32K	32.768kHz from 32KHz internal ULP oscillator
1.024kHz	OSC1K	1.024kHz from 32KHz internal oscillator
32.768kHz	OSC32K	32.768kHz from 32KHz internal oscillator
1.024kHz	XOSC1K	1.024kHz from 32KHz internal oscillator
32.768kHz	XOSC32K	32.768kHz from 32KHz external crystal oscillator

Note: For the calendar to operate correctly, an asynchronous clock of 1Hz should be used.

15.4. Extra Information

For extra information, see [Extra Information for RTC \(CAL\) Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

15.5. Examples

For a list of examples related to this driver, see [Examples for RTC CAL Driver](#).

15.6. API Overview

15.6.1. Structure Definitions

15.6.1.1. Struct `rtc_calendar_alarm_time`

Alarm structure containing time of the alarm and a mask to determine when the alarm will trigger.

Table 15-4. Members

Type	Name	Description
enum rtc_calendar_alarm_mask	mask	Alarm mask to determine on what precision the alarm will match
struct rtc_calendar_time	time	Alarm time

15.6.1.2. Struct `rtc_calendar_config`

Configuration structure for the RTC instance. This structure should be initialized using the [rtc_calendar_get_config_defaults\(\)](#) before any user configurations are set.

Table 15-5. Members

Type	Name	Description
struct rtc_calendar_alarm_time	alarm[]	Alarm values
bool	clear_on_match	If <code>true</code> , clears the clock on alarm match
bool	clock_24h	If <code>true</code> , time is represented in 24 hour mode
enum rtc_calendar_prescaler	prescaler	Input clock prescaler for the RTC module
uint16_t	year_init_value	Initial year for counter value 0

15.6.1.3. Struct `rtc_calendar_events`

Event flags for the [rtc_calendar_enable_events\(\)](#) and [rtc_calendar_disable_events\(\)](#).

Table 15-6. Members

Type	Name	Description
bool	generate_event_on_alarm[]	Generate an output event on an alarm channel match against the RTC count
bool	generate_event_on_overflow	Generate an output event on each overflow of the RTC count
bool	generate_event_on_periodic[]	Generate an output event periodically at a binary division of the RTC counter frequency
bool	generate_event_on_tamper	Generate an output event on every tamper input
bool	on_event_to_tamper	Tamper input event and capture the CLOCK value

15.6.1.4. Struct rtc_calendar_time

Time structure containing the time given by or set to the RTC calendar. The structure uses seven values to give second, minute, hour, PM/AM, day, month, and year. It should be initialized via the [rtc_calendar_get_time_defaults\(\)](#) function before use.

Table 15-7. Members

Type	Name	Description
uint8_t	day	Day value, where day 1 is the first day of the month
uint8_t	hour	Hour value
uint8_t	minute	Minute value
uint8_t	month	Month value, where month 1 is January
bool	pm	PM/AM value, <code>true</code> for PM, or <code>false</code> for AM
uint8_t	second	Second value
uint16_t	year	Year value

15.6.1.5. Struct rtc_tamper_config

The configuration structure for the RTC tamper. This structure should be initialized using the [rtc_tamper_get_config_defaults\(\)](#) before any user configurations are set.

Table 15-8. Members

Type	Name	Description
enum rtc_tamper_active_layer_freq_divider	actl_freq_div	Active layer frequency
bool	bkup_reset_on_tamper	Backup register reset on tamper enable
enum rtc_tamper_debounce_freq_divider	deb_freq_div	Debounce frequency
enum rtc_tamper_debounce_seq	deb_seq	Debounce sequential

Type	Name	Description
bool	dma_tamper_enable	DMA on tamper enable
bool	gp0_enable	General Purpose 0/1 Enable
bool	gp_reset_on_tamper	GP register reset on tamper enable
struct rtc_tamper_input_config	in_cfg[]	Tamper IN configuration

15.6.1.6. Struct `rtc_tamper_input_config`

The configuration structure for tamper INn.

Table 15-9. Members

Type	Name	Description
enum rtc_tamper_input_action	action	Tamper input action
bool	debounce_enable	Debounce enable
enum rtc_tamper_level_sel	level	Tamper level select

15.6.2. Macro Definitions

15.6.2.1. Macro `FEATURE_RTC_PERIODIC_INT`

```
#define FEATURE_RTC_PERIODIC_INT
```

Define port features set according to different device family RTC periodic interval interrupt.

15.6.2.2. Macro `FEATURE_RTC_PRESCALER_OFF`

```
#define FEATURE_RTC_PRESCALER_OFF
```

RTC prescaler is off.

15.6.2.3. Macro `FEATURE_RTC_CLOCK_SELECTION`

```
#define FEATURE_RTC_CLOCK_SELECTION
```

RTC clock selection.

15.6.2.4. Macro `FEATURE_RTC_GENERAL_PURPOSE_REG`

```
#define FEATURE_RTC_GENERAL_PURPOSE_REG
```

General purpose registers.

15.6.2.5. Macro `FEATURE_RTC_TAMPER_DETECTION`

```
#define FEATURE_RTC_TAMPER_DETECTION
```

RTC tamper detection.

15.6.2.6. Macro `RTC_TAMPER_DETECT_EVT`

```
#define RTC_TAMPER_DETECT_EVT
```


RTC tamper input event detection bitmask.

15.6.2.7. Macro `RTC_TAMPER_DETECT_ID0`

```
#define RTC_TAMPER_DETECT_ID0
```

RTC tamper ID0 detection bitmask.

15.6.2.8. Macro `RTC_TAMPER_DETECT_ID1`

```
#define RTC_TAMPER_DETECT_ID1
```

RTC tamper ID1 detection bitmask.

15.6.2.9. Macro `RTC_TAMPER_DETECT_ID2`

```
#define RTC_TAMPER_DETECT_ID2
```

RTC tamper ID2 detection bitmask.

15.6.2.10. Macro `RTC_TAMPER_DETECT_ID3`

```
#define RTC_TAMPER_DETECT_ID3
```

RTC tamper ID3 detection bitmask.

15.6.2.11. Macro `RTC_TAMPER_DETECT_ID4`

```
#define RTC_TAMPER_DETECT_ID4
```

RTC tamper ID4 detection bitmask.

15.6.3. Function Definitions

15.6.3.1. Configuration and Initialization

Function `rtc_calendar_get_time_defaults()`

Initialize a `time` structure.

```
void rtc_calendar_get_time_defaults(  
    struct rtc_calendar_time *const time)
```

This will initialize a given time structure to the time 00:00:00 (hh:mm:ss) and date 2000-01-01 (YYYY-MM-DD).

Table 15-10. Parameters

Data direction	Parameter name	Description
[out]	time	Time structure to initialize

Function `rtc_calendar_get_config_defaults()`

Gets the RTC default settings.

```
void rtc_calendar_get_config_defaults(  
    struct rtc_calendar_config *const config)
```

Initializes the configuration structure to the known default values. This function should be called at the start of any RTC initiation.

The default configuration is as follows:

- Input clock divided by a factor of 1024
- Clear on alarm match off
- Continuously sync clock off
- 12 hour calendar
- Start year 2000 (Year 0 in the counter will be year 2000)
- Events off
- Alarms set to January 1, 2000, 00:00:00
- Alarm will match on second, minute, hour, day, month, and year
- Clock read synchronization is enabled for SAM L22

Table 15-11. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to be initialized to default values

Function `rtc_calendar_reset()`

Resets the RTC module.

```
void rtc_calendar_reset(  
    struct rtc_module *const module)
```

Resets the RTC module to hardware defaults.

Table 15-12. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `rtc_calendar_enable()`

Enables the RTC module.

```
void rtc_calendar_enable(  
    struct rtc_module *const module)
```

Enables the RTC module once it has been configured, ready for use. Most module configuration parameters cannot be altered while the module is enabled.

Table 15-13. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `rtc_calendar_disable()`

Disables the RTC module.

```
void rtc_calendar_disable(  
    struct rtc_module *const module)
```

Disables the RTC module.

Table 15-14. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function rtc_calendar_init()

Initializes the RTC module with given configurations.

```
void rtc_calendar_init(
    struct rtc_module *const module,
    Rtc *const hw,
    const struct rtc_calendar_config *const config)
```

Initializes the module, setting up all given configurations to provide the desired functionality of the RTC.

Table 15-15. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Pointer to hardware instance
[in]	config	Pointer to the configuration structure

Function rtc_calendar_swap_time_mode()

Swaps between 12h and 24h clock mode.

```
void rtc_calendar_swap_time_mode(
    struct rtc_module *const module)
```

Swaps the current RTC time mode:

- If currently in 12h mode, it will swap to 24h
- If currently in 24h mode, it will swap to 12h

Note: This will not change setting in user's configuration structure.

Table 15-16. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function rtc_calendar_frequency_correction()

Calibrate for too-slow or too-fast oscillator.

```
enum status_code rtc_calendar_frequency_correction(
    struct rtc_module *const module,
    const int8_t value)
```

When used, the RTC will compensate for an inaccurate oscillator. The RTC module will add or subtract cycles from the RTC prescaler to adjust the frequency in approximately 1 PPM steps. The provided correction value should be between -127 and 127, allowing for a maximum 127 PPM correction in either direction.

If no correction is needed, set value to zero.

Note: Can only be used when the RTC is operated at 1Hz.

Table 15-17. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	value	Between -127 and 127 used for the correction

Returns

Status of the calibration procedure.

Table 15-18. Return Values

Return value	Description
STATUS_OK	If calibration was done correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

15.6.3.2. Time and Alarm Management

Function `rtc_calendar_time_to_register_value()`

Convert time structure to register_value. Retrieves register_value convert by the time structure.

```
uint32_t rtc_calendar_time_to_register_value(  
    struct rtc_module *const module,  
    const struct rtc_calendar_time *const time)
```

Table 15-19. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	time	Pointer to the time structure

Returns

32-bit value.

Function `rtc_calendar_register_value_to_time()`

Convert register_value to time structure. Retrieves the time structure convert by register_value.

```
void rtc_calendar_register_value_to_time(  
    struct rtc_module *const module,  
    const uint32_t register_value,  
    struct rtc_calendar_time *const time)
```

Table 15-20. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	register_value	The value stored in register
[out]	time	Pointer to the time structure

Function `rtc_calendar_set_time()`

Set the current calendar time to desired time.

```
void rtc_calendar_set_time(  
    struct rtc_module *const module,  
    const struct rtc_calendar_time *const time)
```

Sets the time provided to the calendar.

Table 15-21. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	time	The time to set in the calendar

Function `rtc_calendar_get_time()`

Get the current calendar value.

```
void rtc_calendar_get_time(  
    struct rtc_module *const module,  
    struct rtc_calendar_time *const time)
```

Retrieves the current time of the calendar.

Table 15-22. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	time	Pointer to value that will be filled with current time

Function `rtc_calendar_set_alarm()`

Set the alarm time for the specified alarm.

```
enum status_code rtc_calendar_set_alarm(  
    struct rtc_module *const module,  
    const struct rtc_calendar_alarm_time *const alarm,  
    const enum rtc_calendar_alarm alarm_index)
```

Sets the time and mask specified to the requested alarm.

Table 15-23. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	alarm	The alarm struct to set the alarm with
[in]	alarm_index	The index of the alarm to set

Returns

Status of setting alarm.

Table 15-24. Return Values

Return value	Description
STATUS_OK	If alarm was set correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

Function rtc_calendar_get_alarm()

Get the current alarm time of specified alarm.

```
enum status_code rtc_calendar_get_alarm(
    struct rtc_module *const module,
    struct rtc_calendar_alarm_time *const alarm,
    const enum rtc_calendar_alarm alarm_index)
```

Retrieves the current alarm time for the alarm specified alarm.

Table 15-25. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	alarm	Pointer to the struct that will be filled with alarm time and mask of the specified alarm
[in]	alarm_index	Index of alarm to get alarm time from

Returns

Status of getting alarm.

Table 15-26. Return Values

Return value	Description
STATUS_OK	If alarm was read correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

15.6.3.3. Status Flag Management**Function rtc_calendar_is_overflow()**

Check if an RTC overflow has occurred.

```
bool rtc_calendar_is_overflow(
    struct rtc_module *const module)
```

Checks the overflow flag in the RTC. The flag is set when there is an overflow in the clock.

Table 15-27. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Returns

Overflow state of the RTC module.

Table 15-28. Return Values

Return value	Description
true	If the RTC count value has overflowed
false	If the RTC count value has not overflowed

Function `rtc_calendar_clear_overflow()`

Clears the RTC overflow flag.

```
void rtc_calendar_clear_overflow(  
    struct rtc_module *const module)
```

Table 15-29. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Clears the RTC module counter overflow flag, so that new overflow conditions can be detected.

Function `rtc_calendar_is_periodic_interval()`

Check if an RTC periodic interval interrupt has occurred.

```
bool rtc_calendar_is_periodic_interval(  
    struct rtc_module *const module,  
    enum rtc_calendar_periodic_interval n)
```

Checks the periodic interval flag in the RTC.

Table 15-30. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	n	RTC periodic interval interrupt

Returns

Periodic interval interrupt state of the RTC module.

Table 15-31. Return Values

Return value	Description
true	RTC periodic interval interrupt occur
false	RTC periodic interval interrupt doesn't occur

Function `rtc_calendar_clear_periodic_interval()`

Clears the RTC periodic interval flag.

```
void rtc_calendar_clear_periodic_interval(  
    struct rtc_module *const module,  
    enum rtc_calendar_periodic_interval n)
```

Clears the RTC module counter periodic interval flag, so that new periodic interval conditions can be detected.

Table 15-32. Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	n	RTC periodic interval interrupt

Function `rtc_calendar_is_alarm_match()`

Check the RTC alarm flag.

```
bool rtc_calendar_is_alarm_match(  
    struct rtc_module *const module,  
    const enum rtc_calendar_alarm alarm_index)
```

Check if the specified alarm flag is set. The flag is set when there is a compare match between the alarm value and the clock.

Table 15-33. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	alarm_index	Index of the alarm to check

Returns

Match status of the specified alarm.

Table 15-34. Return Values

Return value	Description
true	If the specified alarm has matched the current time
false	If the specified alarm has not matched the current time

Function `rtc_calendar_clear_alarm_match()`

Clears the RTC alarm match flag.

```
enum status_code rtc_calendar_clear_alarm_match(  
    struct rtc_module *const module,  
    const enum rtc_calendar_alarm alarm_index)
```

Clear the requested alarm match flag, so that future alarm matches can be determined.

Table 15-35. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	alarm_index	The index of the alarm match to clear

Returns

Status of the alarm match clear operation.

Table 15-36. Return Values

Return value	Description
STATUS_OK	If flag was cleared correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

15.6.3.4. Event Management**Function `rtc_calendar_enable_events()`**

Enables an RTC event output.

```
void rtc_calendar_enable_events(
    struct rtc_module *const module,
    struct rtc_calendar_events *const events)
```

Enables one or more output events from the RTC module. See [rtc_calendar_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 15-37. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	events	Struct containing flags of events to enable

Function `rtc_calendar_disable_events()`

Disables an RTC event output.

```
void rtc_calendar_disable_events(
    struct rtc_module *const module,
    struct rtc_calendar_events *const events)
```

Disabled one or more output events from the RTC module. See [rtc_calendar_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 15-38. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	events	Struct containing flags of events to disable

15.6.3.5. RTC General Purpose Registers

Function `rtc_write_general_purpose_reg()`

Write a value into general purpose register.

```
void rtc_write_general_purpose_reg(
    struct rtc_module *const module,
    const uint8_t index,
    uint32_t value)
```

Table 15-39. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	n	General purpose type
[in]	index	General purpose register index (0..3)

Function `rtc_read_general_purpose_reg()`

Read the value from general purpose register.

```
uint32_t rtc_read_general_purpose_reg(
    struct rtc_module *const module,
    const uint8_t index)
```

Table 15-40. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	index	General purpose register index (0..3)

Returns

Value of general purpose register.

15.6.3.6. Callbacks

Function `rtc_calendar_register_callback()`

Registers callback for the specified callback type.

```
enum status_code rtc_calendar_register_callback(
    struct rtc_module *const module,
    rtc_calendar_callback_t callback,
    enum rtc_calendar_callback callback_type)
```

Associates the given callback function with the specified callback type. To enable the callback, the [rtc_calendar_enable_callback](#) function must be used.

Table 15-41. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback	Pointer to the function desired for the specified callback
[in]	callback_type	Callback type to register

Returns

Status of registering callback.

Table 15-42. Return Values

Return value	Description
STATUS_OK	Registering was done successfully
STATUS_ERR_INVALID_ARG	If trying to register, a callback is not available

Function rtc_calendar_unregister_callback()

Unregisters callback for the specified callback type.

```
enum status_code rtc_calendar_unregister_callback(
    struct rtc_module *const module,
    enum rtc_calendar_callback callback_type)
```

When called, the currently registered callback for the given callback type will be removed.

Table 15-43. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Specifies the callback type to unregister

Returns

Status of unregistering callback.

Table 15-44. Return Values

Return value	Description
STATUS_OK	Unregistering was done successfully
STATUS_ERR_INVALID_ARG	If trying to unregister, a callback is not available

Function rtc_calendar_enable_callback()

Enables callback.

```
void rtc_calendar_enable_callback(
    struct rtc_module *const module,
    enum rtc_calendar_callback callback_type)
```

Enables the callback specified by the callback_type.

Table 15-45. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Callback type to enable

Function rtc_calendar_disable_callback()

Disables callback.

```
void rtc_calendar_disable_callback(
    struct rtc_module *const module,
    enum rtc_calendar_callback callback_type)
```

Disables the callback specified by the callback_type.

Table 15-46. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Callback type to disable

15.6.3.7. RTC Tamper Detection**Function rtc_tamper_get_config_defaults()**

Gets the RTC tamper default configurations.

```
void rtc_tamper_get_config_defaults(
    struct rtc_tamper_config *const config)
```

Initializes the configuration structure to default values.

The default configuration is as follows:

- Disable backup register reset on tamper
- Disable GP register reset on tamper
- Active layer clock divided by a factor of 8
- Debounce clock divided by a factor of 8
- Detect edge on INn with synchronous stability debouncing
- Disable DMA on tamper
- Enable GP register
- Disable debounce, detect on falling edge and no action on INn

Table 15-47. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to be initialized to default values.

Function rtc_tamper_set_config()

```
enum status_code rtc_tamper_set_config(
    struct rtc_module *const module,
    struct rtc_tamper_config *const tamper_cfg)
```

Function `rtc_tamper_get_detect_flag()`

Retrieves the RTC tamper detection status.

```
uint32_t rtc_tamper_get_detect_flag(  
    struct rtc_module *const module)
```

Retrieves the detection status of each input pin and the input event.

Table 15-48. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the RTC software instance struct

Returns

Bitmask of detection flags.

Table 15-49. Return Values

Return value	Description
RTC_TAMPER_DETECT_ID0	Tamper condition on IN0 has been detected
RTC_TAMPER_DETECT_ID1	Tamper condition on IN1 has been detected
RTC_TAMPER_DETECT_ID2	Tamper condition on IN2 has been detected
RTC_TAMPER_DETECT_ID3	Tamper condition on IN3 has been detected
RTC_TAMPER_DETECT_ID4	Tamper condition on IN4 has been detected
RTC_TAMPER_DETECT_EVT	Tamper input event has been detected

Function `rtc_tamper_clear_detect_flag()`

Clears RTC tamper detection flag.

```
void rtc_tamper_clear_detect_flag(  
    struct rtc_module *const module,  
    const uint32_t detect_flags)
```

Clears the given detection flag of the module.

Table 15-50. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the TC software instance struct
[in]	detect_flags	Bitmask of detection flags

15.6.3.8. Function `rtc_tamper_get_stamp()`

Get the tamper stamp value.

```
void rtc_tamper_get_stamp(  
    struct rtc_module *const module,  
    struct rtc_calendar_time *const time)
```

Table 15-51. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	time	Pointer to value that filled with tamper stamp time

15.6.4. Enumeration Definitions

15.6.4.1. Enum rtc_calendar_alarm

Available alarm channels.

Note: Not all alarm channels are available on all devices.

Table 15-52. Members

Enum value	Description
RTC_CALENDAR_ALARM_0	Alarm channel 0
RTC_CALENDAR_ALARM_1	Alarm channel 1
RTC_CALENDAR_ALARM_2	Alarm channel 2
RTC_CALENDAR_ALARM_3	Alarm channel 3

15.6.4.2. Enum rtc_calendar_alarm_mask

Available mask options for alarms.

Table 15-53. Members

Enum value	Description
RTC_CALENDAR_ALARM_MASK_DISABLED	Alarm disabled
RTC_CALENDAR_ALARM_MASK_SEC	Alarm match on second
RTC_CALENDAR_ALARM_MASK_MIN	Alarm match on second and minute
RTC_CALENDAR_ALARM_MASK_HOUR	Alarm match on second, minute, and hour
RTC_CALENDAR_ALARM_MASK_DAY	Alarm match on second, minute, hour, and day
RTC_CALENDAR_ALARM_MASK_MONTH	Alarm match on second, minute, hour, day, and month
RTC_CALENDAR_ALARM_MASK_YEAR	Alarm match on second, minute, hour, day, month, and year

15.6.4.3. Enum rtc_calendar_callback

The available callback types for the RTC calendar module.

Table 15-54. Members

Enum value	Description
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_0	Callback for Periodic Interval 0 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_1	Callback for Periodic Interval 1 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_2	Callback for Periodic Interval 2 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_3	Callback for Periodic Interval 3 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_4	Callback for Periodic Interval 4 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_5	Callback for Periodic Interval 5 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_6	Callback for Periodic Interval 6 Interrupt
RTC_CALENDAR_CALLBACK_PERIODIC_INTERVAL_7	Callback for Periodic Interval 7 Interrupt
RTC_CALENDAR_CALLBACK_ALARM_0	Callback for alarm 0
RTC_CALENDAR_CALLBACK_ALARM_1	Callback for alarm 1
RTC_CALENDAR_CALLBACK_ALARM_2	Callback for alarm 2
RTC_CALENDAR_CALLBACK_ALARM_3	Callback for alarm 3
RTC_CALENDAR_CALLBACK_TAMPER	Callback for tamper
RTC_CALENDAR_CALLBACK_OVERFLOW	Callback for overflow

15.6.4.4. Enum `rtc_calendar_periodic_interval`

Table 15-55. Members

Enum value	Description
RTC_CALENDAR_PERIODIC_INTERVAL_0	Periodic interval 0
RTC_CALENDAR_PERIODIC_INTERVAL_1	Periodic interval 1
RTC_CALENDAR_PERIODIC_INTERVAL_2	Periodic interval 2
RTC_CALENDAR_PERIODIC_INTERVAL_3	Periodic interval 3
RTC_CALENDAR_PERIODIC_INTERVAL_4	Periodic interval 4
RTC_CALENDAR_PERIODIC_INTERVAL_5	Periodic interval 5
RTC_CALENDAR_PERIODIC_INTERVAL_6	Periodic interval 6
RTC_CALENDAR_PERIODIC_INTERVAL_7	Periodic interval 7

15.6.4.5. Enum `rtc_calendar_prescaler`

The available input clock prescaler values for the RTC calendar module.

Table 15-56. Members

Enum value	Description
RTC_CALENDAR_PRESCALER_OFF	RTC prescaler is off, and the input clock frequency is prescaled by a factor of 1
RTC_CALENDAR_PRESCALER_DIV_1	RTC input clock frequency is prescaled by a factor of 1
RTC_CALENDAR_PRESCALER_DIV_2	RTC input clock frequency is prescaled by a factor of 2
RTC_CALENDAR_PRESCALER_DIV_4	RTC input clock frequency is prescaled by a factor of 4
RTC_CALENDAR_PRESCALER_DIV_8	RTC input clock frequency is prescaled by a factor of 8
RTC_CALENDAR_PRESCALER_DIV_16	RTC input clock frequency is prescaled by a factor of 16
RTC_CALENDAR_PRESCALER_DIV_32	RTC input clock frequency is prescaled by a factor of 32
RTC_CALENDAR_PRESCALER_DIV_64	RTC input clock frequency is prescaled by a factor of 64
RTC_CALENDAR_PRESCALER_DIV_128	RTC input clock frequency is prescaled by a factor of 128
RTC_CALENDAR_PRESCALER_DIV_256	RTC input clock frequency is prescaled by a factor of 256
RTC_CALENDAR_PRESCALER_DIV_512	RTC input clock frequency is prescaled by a factor of 512
RTC_CALENDAR_PRESCALER_DIV_1024	RTC input clock frequency is prescaled by a factor of 1024

15.6.4.6. Enum `rtc_clock_sel`

Table 15-57. Members

Enum value	Description
RTC_CLOCK_SELECTION_ULP1K	1.024kHz from 32KHz internal ULP oscillator
RTC_CLOCK_SELECTION_ULP32K	32.768kHz from 32KHz internal ULP oscillator
RTC_CLOCK_SELECTION_OSC1K	1.024kHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_OSC32K	32.768kHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_XOSC1K	1.024kHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_XOSC32K	32.768kHz from 32.768kHz external crystal oscillator

15.6.4.7. Enum `rtc_tamper_active_layer_freq_divider`

The available prescaler factor for the RTC clock output used during active layer protection.

Table 15-58. Members

Enum value	Description
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_2	RTC active layer frequency is prescaled by a factor of 2
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_4	RTC active layer frequency is prescaled by a factor of 4
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_8	RTC active layer frequency is prescaled by a factor of 8
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_16	RTC active layer frequency is prescaled by a factor of 16
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_32	RTC active layer frequency is prescaled by a factor of 32
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_64	RTC active layer frequency is prescaled by a factor of 64
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_128	RTC active layer frequency is prescaled by a factor of 128
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_256	RTC active layer frequency is prescaled by a factor of 256

15.6.4.8. Enum rtc_tamper_debounce_freq_divider

The available prescaler factor for the input debouncers.

Table 15-59. Members

Enum value	Description
RTC_TAMPER_DEBOUNCE_FREQ_DIV_2	RTC debounce frequency is prescaled by a factor of 2
RTC_TAMPER_DEBOUNCE_FREQ_DIV_4	RTC debounce frequency is prescaled by a factor of 4
RTC_TAMPER_DEBOUNCE_FREQ_DIV_8	RTC debounce frequency is prescaled by a factor of 8
RTC_TAMPER_DEBOUNCE_FREQ_DIV_16	RTC debounce frequency is prescaled by a factor of 16
RTC_TAMPER_DEBOUNCE_FREQ_DIV_32	RTC debounce frequency is prescaled by a factor of 32
RTC_TAMPER_DEBOUNCE_FREQ_DIV_64	RTC debounce frequency is prescaled by a factor of 64
RTC_TAMPER_DEBOUNCE_FREQ_DIV_128	RTC debounce frequency is prescaled by a factor of 128
RTC_TAMPER_DEBOUNCE_FREQ_DIV_256	RTC debounce frequency is prescaled by a factor of 256

15.6.4.9. Enum rtc_tamper_debounce_seq

The available sequential for tamper debounce.

Table 15-60. Members

Enum value	Description
RTC_TAMPER_DEBOUNCE_SYNC	Tamper input detect edge with synchronous stability debounce
RTC_TAMPER_DEBOUNCE_ASYNC	Tamper input detect edge with asynchronous stability debounce
RTC_TAMPER_DEBOUNCE_MAJORITY	Tamper input detect edge with majority debounce

15.6.4.10. Enum rtc_tamper_input_action

The available action taken by the tamper input.

Table 15-61. Members

Enum value	Description
RTC_TAMPER_INPUT_ACTION_OFF	RTC tamper input action is disabled
RTC_TAMPER_INPUT_ACTION_WAKE	RTC tamper input action is wake and set tamper flag
RTC_TAMPER_INPUT_ACTION_CAPTURE	RTC tamper input action is capture timestamp and set tamper flag
RTC_TAMPER_INPUT_ACTION_ACTL	RTC tamper input action is compare IN to OUT, when a mismatch occurs, capture timestamp and set tamper flag

15.6.4.11. Enum rtc_tamper_level_sel

The available edge condition for tamper INn level select.

Table 15-62. Members

Enum value	Description
RTC_TAMPER_LEVEL_FALLING	A falling edge condition will be detected on Tamper input
RTC_TAMPER_LEVEL_RISING	A rising edge condition will be detected on Tamper input

15.7. RTC Tamper Detect

The RTC provides several selectable polarity external inputs (INn) that can be used for tamper detection. When detect, tamper inputs support the four actions:

- Off
- Wake
- Capture
- Active layer protection

Note: The Active Layer Protection is a means of detecting broken traces on the PCB provided by RTC. In this mode an RTC output signal is routed over critical components on the board and fed back to one of the RTC inputs. The input and output signals are compared and a tamper condition is detected when they do not match.

Separate debouncers are embedded for each external input. The detection time depends on whether the debouncer operates synchronously or asynchronously, and whether majority detection is enabled or not. For details, refer to the section "Tamper Detection" of datasheet.

15.8. Extra Information for RTC (CAL) Driver

15.8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
RTC	Real Time Counter
PPM	Part Per Million
RC	Resistor/Capacitor

15.8.2. Dependencies

This driver has the following dependencies:

- None

15.8.3. Errata

There are no errata related to this driver.

15.8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added support for RTC tamper feature
Added driver instance parameter to all API function calls, except get_config_defaults
Updated initialization function to also enable the digital interface clock to the module if it is disabled
Initial release

15.9. Examples for RTC CAL Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM RTC Calendar \(RTC CAL\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for RTC \(CAL\) - Basic](#)
- [Quick Start Guide for RTC \(CAL\) - Callback](#)

15.9.1. Quick Start Guide for RTC (CAL) - Basic

In this use case, the RTC is set up in calendar mode. The time is set and also an alarm is set to show a general use of the RTC in calendar mode. Also the clock is swapped from 24h to 12h mode after initialization. The board LED will be toggled once the current time matches the set time.

15.9.1.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

Clocks and Oscillators

The `conf_clock.h` file needs to be changed with the following values to configure the clocks and oscillators for the module.

The following oscillator settings are needed:

```
/* SYSTEM_CLOCK_SOURCE_OSC32K configuration - Internal 32KHz oscillator */
# define CONF_CLOCK_OSC32K_ENABLE true
# define CONF_CLOCK_OSC32K_STARTUP_TIME SYSTEM_OSC32K_STARTUP_130
# define CONF_CLOCK_OSC32K_ENABLE_1KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ENABLE_32KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ON_DEMAND true
# define CONF_CLOCK_OSC32K_RUN_IN_STANDBY false
```

The following generic clock settings are needed:

```
/* Configure GCLK generator 2 (RTC) */
# define CONF_CLOCK_GCLK_2_ENABLE true
# define CONF_CLOCK_GCLK_2_RUN_IN_STANDBY false
# define CONF_CLOCK_GCLK_2_CLOCK_SOURCE SYSTEM_CLOCK_SOURCE_OSC32K
# define CONF_CLOCK_GCLK_2_PRESCALER 32
# define CONF_CLOCK_GCLK_2_OUTPUT_ENABLE false
```

15.9.1.2. Setup

Initialization Code

Create an `rtc_module` struct and add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

Copy-paste the following setup code to your application:

```
void configure_rtc_calendar(void)
{
    /* Initialize RTC in calendar mode. */
    struct rtc_calendar_config config_rtc_calendar;
    rtc_calendar_get_config_defaults(&config_rtc_calendar);

    struct rtc_calendar_time alarm;
    rtc_calendar_get_time_defaults(&alarm);
    alarm.year = 2013;
    alarm.month = 1;
    alarm.day = 1;
    alarm.hour = 0;
    alarm.minute = 0;
    alarm.second = 4;

    config_rtc_calendar.clock_24h = true;
    config_rtc_calendar.alarm[0].time = alarm;
    config_rtc_calendar.alarm[0].mask = RTC_CALENDAR_ALARM_MASK_YEAR;
```

```

    rtc_calendar_init(&rtc_instance, RTC, &config_rtc_calendar);
    rtc_calendar_enable(&rtc_instance);
}

```

Add to Main

Add the following to `main()`.

```

system_init();

struct rtc_calendar_time time;
time.year = 2012;
time.month = 12;
time.day = 31;
time.hour = 23;
time.minute = 59;
time.second = 59;

configure_rtc_calendar();

/* Set current time. */
rtc_calendar_set_time(&rtc_instance, &time);

rtc_calendar_swap_time_mode(&rtc_instance);

```

Workflow

1. Make configuration structure.

```
struct rtc_calendar_config config_rtc_calendar;
```

2. Fill the configuration structure with the default driver configuration.

```
rtc_calendar_get_config_defaults(&config_rtc_calendar);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Make time structure for alarm and set with default and desired values.

```

struct rtc_calendar_time alarm;
rtc_calendar_get_time_defaults(&alarm);
alarm.year = 2013;
alarm.month = 1;
alarm.day = 1;
alarm.hour = 0;
alarm.minute = 0;
alarm.second = 4;

```

4. Change configurations as desired.

```

config_rtc_calendar.clock_24h = true;
config_rtc_calendar.alarm[0].time = alarm;
config_rtc_calendar.alarm[0].mask = RTC_CALENDAR_ALARM_MASK_YEAR;

```

5. Initialize module.

```
rtc_calendar_init(&rtc_instance, RTC, &config_rtc_calendar);
```

6. Enable module.

```
rtc_calendar_enable(&rtc_instance);
```

15.9.1.3. Implementation

Add the following to `main()`.

```
while (true) {
    if (rtc_calendar_is_alarm_match(&rtc_instance, RTC_CALENDAR_ALARM_0)) {
        /* Do something on RTC alarm match here */
        port_pin_toggle_output_level(LED_0_PIN);

        rtc_calendar_clear_alarm_match(&rtc_instance,
        RTC_CALENDAR_ALARM_0);
    }
}
```

Workflow

1. Start an infinite loop, to continuously poll for a RTC alarm match.

```
while (true) {
```

2. Check to see if a RTC alarm match has occurred.

```
if (rtc_calendar_is_alarm_match(&rtc_instance, RTC_CALENDAR_ALARM_0)) {
```

3. Once an alarm match occurs, perform the desired user action.

```
/* Do something on RTC alarm match here */
port_pin_toggle_output_level(LED_0_PIN);
```

4. Clear the alarm match, so that future alarms may occur.

```
rtc_calendar_clear_alarm_match(&rtc_instance, RTC_CALENDAR_ALARM_0);
```

15.9.2. Quick Start Guide for RTC (CAL) - Callback

In this use case, the RTC is set up in calendar mode. The time is set and an alarm is enabled, as well as a callback for when the alarm time is hit. Each time the callback fires, the alarm time is reset to five seconds in the future and the board LED toggled.

15.9.2.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

Clocks and Oscillators

The `conf_clock.h` file needs to be changed with the following values to configure the clocks and oscillators for the module.

The following oscillator settings are needed:

```
/* SYSTEM_CLOCK_SOURCE_OSC32K configuration - Internal 32KHz oscillator */
# define CONF_CLOCK_OSC32K_ENABLE true
# define CONF_CLOCK_OSC32K_STARTUP_TIME SYSTEM_OSC32K_STARTUP_130
# define CONF_CLOCK_OSC32K_ENABLE_1KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ENABLE_32KHZ_OUTPUT true
# define CONF_CLOCK_OSC32K_ON_DEMAND true
# define CONF_CLOCK_OSC32K_RUN_IN_STANDBY false
```

The following generic clock settings are needed:

```
/* Configure GCLK generator 2 (RTC) */
# define CONF_CLOCK_GCLK_2_ENABLE true
# define CONF_CLOCK_GCLK_2_RUN_IN_STANDBY false
# define CONF_CLOCK_GCLK_2_CLOCK_SOURCE
```

```

SYSTEM_CLOCK_SOURCE_OSC32K
# define CONF_CLOCK_GCLK_2_PRESCALER          32
# define CONF_CLOCK_GCLK_2_OUTPUT_ENABLE      false

```

15.9.2.2. Setup

Code

Create an `rtc_module` struct and add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

The following must be added to the user application:

Function for setting up the module:

```

void configure_rtc_calendar(void)
{
    /* Initialize RTC in calendar mode. */
    struct rtc_calendar_config config_rtc_calendar;
    rtc_calendar_get_config_defaults(&config_rtc_calendar);

    alarm.time.year      = 2013;
    alarm.time.month     = 1;
    alarm.time.day       = 1;
    alarm.time.hour      = 0;
    alarm.time.minute    = 0;
    alarm.time.second     = 4;

    config_rtc_calendar.clock_24h = true;
    config_rtc_calendar.alarm[0].time = alarm.time;
    config_rtc_calendar.alarm[0].mask = RTC_CALENDAR_ALARM_MASK_YEAR;

    rtc_calendar_init(&rtc_instance, RTC, &config_rtc_calendar);

    rtc_calendar_enable(&rtc_instance);
}

```

Callback function:

```

void rtc_match_callback(void)
{
    /* Do something on RTC alarm match here */
    port_pin_toggle_output_level(LED_0_PIN);

    /* Set new alarm in 5 seconds */
    alarm.mask = RTC_CALENDAR_ALARM_MASK_SEC;

    alarm.time.second += 5;
    alarm.time.second = alarm.time.second % 60;

    rtc_calendar_set_alarm(&rtc_instance, &alarm, RTC_CALENDAR_ALARM_0);
}

```

Function for setting up the callback functionality of the driver:

```

void configure_rtc_callbacks(void)
{
    rtc_calendar_register_callback(
        &rtc_instance, rtc_match_callback,
        RTC_CALENDAR_CALLBACK_ALARM_0);
    rtc_calendar_enable_callback(&rtc_instance,
        RTC_CALENDAR_CALLBACK_ALARM_0);
}

```

Add to user application `main()`:

```
system_init();

struct rtc_calendar_time time;
rtc_calendar_get_time_defaults(&time);
time.year    = 2012;
time.month   = 12;
time.day      = 31;
time.hour     = 23;
time.minute  = 59;
time.second  = 59;

/* Configure and enable RTC */
configure_rtc_calendar();

/* Configure and enable callback */
configure_rtc_callbacks();

/* Set current time. */
rtc_calendar_set_time(&rtc_instance, &time);
```

Workflow

1. Initialize system.

```
system_init();
```

2. Create and initialize a time structure.

```
struct rtc_calendar_time time;
rtc_calendar_get_time_defaults(&time);
time.year    = 2012;
time.month   = 12;
time.day      = 31;
time.hour     = 23;
time.minute  = 59;
time.second  = 59;
```

3. Configure and enable module.

```
configure_rtc_calendar();
```

1. Create an RTC configuration structure to hold the desired RTC driver settings and fill it with the default driver configuration values.

```
struct rtc_calendar_config config_rtc_calendar;
rtc_calendar_get_config_defaults(&config_rtc_calendar);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

2. Create and initialize an alarm.

```
alarm.time.year    = 2013;
alarm.time.month    = 1;
alarm.time.day      = 1;
alarm.time.hour     = 0;
alarm.time.minute   = 0;
alarm.time.second   = 4;
```


3. Change settings in the configuration and set alarm.

```
config_rtc_calendar.clock_24h = true;
config_rtc_calendar.alarm[0].time = alarm.time;
config_rtc_calendar.alarm[0].mask = RTC_CALENDAR_ALARM_MASK_YEAR;
```

4. Initialize the module with the set configurations.

```
rtc_calendar_init(&rtc_instance, RTC, &config_rtc_calendar);
```

5. Enable the module.

```
rtc_calendar_enable(&rtc_instance);
```

4. Configure callback functionality.

```
configure_rtc_callbacks();
```

1. Register overflow callback.

```
rtc_calendar_register_callback(
    &rtc_instance, rtc_match_callback,
    RTC_CALENDAR_CALLBACK_ALARM_0);
```

2. Enable overflow callback.

```
rtc_calendar_enable_callback(&rtc_instance,
    RTC_CALENDAR_CALLBACK_ALARM_0);
```

5. Set time of the RTC calendar.

```
rtc_calendar_set_time(&rtc_instance, &time);
```

15.9.2.3. Implementation

Code

Add to user application main:

```
while (true) {
    /* Infinite loop */
}
```

Workflow

1. Infinite while loop while waiting for callbacks.

```
while (true) {
```

15.9.2.4. Callback

Each time the RTC time matches the configured alarm, the callback function will be called.

Workflow

1. Create alarm struct and initialize the time with current time.

```
struct rtc_calendar_alarm_time alarm;
```

2. Set alarm to trigger on seconds only.

```
alarm.mask = RTC_CALENDAR_ALARM_MASK_SEC;
```

3. Add one second to the current time and set new alarm.

```
alarm.time.second += 5;
alarm.time.second = alarm.time.second % 60;
```

```
rtc_calendar_set_alarm(&rtc_instance, &alarm, RTC_CALENDAR_ALARM_0);
```

16. SAM Serial USART (SERCOM USART) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the SERCOM module in its USART mode to transfer or receive USART data frames. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

16.1. Prerequisites

To use the USART you need to have a GCLK generator enabled and running that can be used as the SERCOM clock source. This can either be configured in `conf_clocks.h` or by using the system clock driver.

16.2. Module Overview

This driver will use one (or more) SERCOM interface(s) in the system and configure it to run as a USART interface in either synchronous or asynchronous mode.

16.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_USART_SYNC_SCHEME_V2	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_OVER_SAMPLE	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_HARDWARE_FLOW_CONTROL	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_IRDA	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_LIN_SLAVE	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_COLLISION_DECTION	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_START_FRAME_DECTION	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_IMMEDIATE_BUFFER_OVERFLOW_NOTIFICATION	SAM D21/R21/D09/D10/D11/L21/ L22/DA1/C20/C21
FEATURE_USART_RS485	SAM C20/C21
FEATURE_USART_LIN_MASTER	SAM L22/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

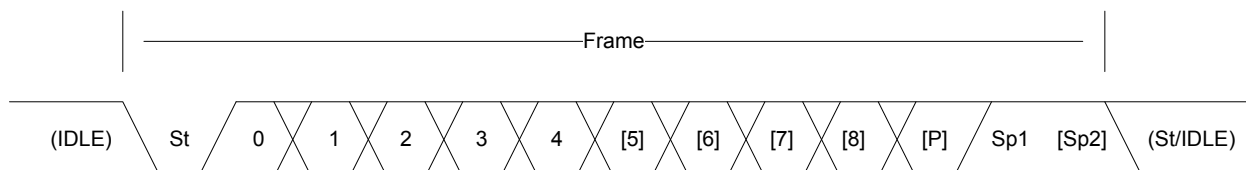
16.2.2. Frame Format

Communication is based on frames, where the frame format can be customized to accommodate a wide range of standards. A frame consists of a start bit, a number of data bits, an optional parity bit for error detection as well as a configurable length stop bit(s) - see [Figure 16-1](#). [Table 16-1](#) shows the available parameters you can change in a frame.

Table 16-1. USART Frame Parameters

Parameter	Options
Start bit	1
Data bits	5, 6, 7, 8, 9
Parity bit	None, Even, Odd
Stop bits	1, 2

Figure 16-1. USART Frame Overview



16.2.3. Synchronous Mode

In synchronous mode a dedicated clock line is provided; either by the USART itself if in master mode, or by an external master if in slave mode. Maximum transmission speed is the same as the GCLK clocking the USART peripheral when in slave mode, and the GCLK divided by two if in master mode. In synchronous mode the interface needs three lines to communicate:

- TX (Transmit pin)
- RX (Receive pin)
- XCK (Clock pin)

16.2.3.1. Data Sampling

In synchronous mode the data is sampled on either the rising or falling edge of the clock signal. This is configured by setting the clock polarity in the configuration struct.

16.2.4. Asynchronous Mode

In asynchronous mode no dedicated clock line is used, and the communication is based on matching the clock speed on the transmitter and receiver. The clock is generated from the internal SERCOM baudrate generator, and the frames are synchronized by using the frame start bits. Maximum transmission speed is limited to the SERCOM GCLK divided by 16. In asynchronous mode the interface only needs two lines to communicate:

- TX (Transmit pin)
- RX (Receive pin)

16.2.4.1. Transmitter/receiver Clock Matching

For successful transmit and receive using the asynchronous mode the receiver and transmitter clocks needs to be closely matched. When receiving a frame that does not match the selected baudrate closely enough the receiver will be unable to synchronize the frame(s), and garbage transmissions will result.

16.2.5. Parity

Parity can be enabled to detect if a transmission was in error. This is done by counting the number of "1" bits in the frame. When using even parity the parity bit will be set if the total number of "1"s in the frame are an even number. If using odd parity the parity bit will be set if the total number of "1"s are odd.

When receiving a character the receiver will count the number of "1"s in the frame and give an error if the received frame and parity bit disagree.

16.2.6. GPIO Configuration

The SERCOM module has four internal pads; the RX pin can be placed freely on any one of the four pads, and the TX and XCK pins have two predefined positions that can be selected as a pair. The pads can then be routed to an external GPIO pin using the normal pin multiplexing scheme on the SAM.

16.3. Special Considerations

Never execute large portions of code in the callbacks. These are run from the interrupt routine, and thus having long callbacks will keep the processor in the interrupt handler for an equally long time. A common way to handle this is to use global flags signaling the main application that an interrupt event has happened, and only do the minimal needed processing in the callback.

16.4. Extra Information

For extra information, see [Extra Information for SERCOM USART Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

16.5. Examples

For a list of examples related to this driver, see [Examples for SERCOM USART Driver](#).

16.6. API Overview

16.6.1. Variable and Type Definitions

16.6.1.1. Type `usart_callback_t`

```
typedef void(* usart_callback_t )(struct usart_module *const module)
```

Type of the callback functions.

16.6.2. Structure Definitions

16.6.2.1. Struct `iso7816_config_t`

ISO7816 configuration structure.

Table 16-2. Members

Type	Name	Description
bool	enable_inverse	Enable inverse transmission and reception
bool	enabled	
enum iso7816_guard_time	guard_time	Guard time, which lasts two bit times
enum iso7816_inhibit_nack	inhibit_nack	Inhibit Non Acknowledge: 0: the NACK is generated; 1: the NACK is not generated.
uint32_t	max_iterations	
enum iso7816_protocol_type	protocol_t	ISO7816 protocol type
enum iso7816_successive_recv_nack	successive_recv_nack	Disable successive NACKs. 0: NACK is sent on the ISO line as soon as a parity error occurs in the received character. Successive parity errors are counted up to the value in the max_iterations field. These parity errors generate a NACK on the ISO line. As soon as this value is reached, no additional NACK is sent on the ISO line. The ITERATION flag is asserted.

16.6.2.2. Struct `usart_config`

Configuration options for USART.

Table 16-3. Members

Type	Name	Description
uint32_t	baudrate	USART baudrate
enum usart_character_size	character_size	USART character size
bool	clock_polarity_inverted	USART Clock Polarity. If true, data changes on falling XCK edge and is sampled at rising edge. If false, data changes on rising XCK edge and is sampled at falling edge.
bool	collision_detection_enable	Enable collision detection
enum usart_dataorder	data_order	USART bit order (MSB or LSB first)
bool	encoding_format_enable	Enable IrDA encoding format

Type	Name	Description
uint32_t	ext_clock_freq	External clock frequency in synchronous mode. This must be set if <code>use_external_clock</code> is true.
enum gclk_generator	generator_source	GCLK generator source
bool	immediate_buffer_overflow_notification	Controls when the buffer overflow status bit is asserted when a buffer overflow occurs
struct iso7816_config_t	iso7816_config	Enable ISO7816 for smart card interfacing
enum lin_master_break_length	lin_break_length	LIN Master Break Length
enum lin_master_header_delay	lin_header_delay	LIN master header delay
enum lin_node_type	lin_node	LIN node type
bool	lin_slave_enable	Enable LIN Slave Support
enum usart_signal_mux_settings	mux_setting	USART pin out
enum usart_parity	parity	USART parity
uint32_t	pinmux_pad0	PAD0 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD0, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD0 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD0.
uint32_t	pinmux_pad1	PAD1 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD1, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD1 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD1.

Type	Name	Description
uint32_t	pinmux_pad2	PAD2 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD2, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD2 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD2.
uint32_t	pinmux_pad3	PAD3 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD3, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD3 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD3.
uint8_t	receive_pulse_length	The minimum pulse length required for a pulse to be accepted by the IrDA receiver
bool	receiver_enable	Enable receiver
enum rs485_guard_time	rs485_guard_time	RS485 guard time
bool	run_in_standby	If true the USART will be kept running in Standby sleep mode
enum usart_sample_adjustment	sample_adjustment	USART sample adjustment
enum usart_sample_rate	sample_rate	USART sample rate
bool	start_frame_detection_enable	Enable start of frame detection
enum usart_stopbits	stopbits	Number of stop bits
enum usart_transfer_mode	transfer_mode	USART in asynchronous or synchronous mode

Type	Name	Description
bool	transmitter_enable	Enable transmitter
bool	use_external_clock	States whether to use the external clock applied to the XCK pin. In synchronous mode the shift register will act directly on the XCK clock. In asynchronous mode the XCK will be the input to the USART hardware module.

16.6.2.3. Struct `usart_module`

SERCOM USART driver software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

16.6.3. Macro Definitions

16.6.3.1. Driver Feature Definition

Define SERCOM USART features set according to different device family.

Macro `FEATURE_USART_SYNC_SCHEME_V2`

```
#define FEATURE_USART_SYNC_SCHEME_V2
```

USART sync scheme version 2.

Macro `FEATURE_USART_OVER_SAMPLE`

```
#define FEATURE_USART_OVER_SAMPLE
```

USART oversampling.

Macro `FEATURE_USART_HARDWARE_FLOW_CONTROL`

```
#define FEATURE_USART_HARDWARE_FLOW_CONTROL
```

USART hardware control flow.

Macro `FEATURE_USART_IRDA`

```
#define FEATURE_USART_IRDA
```

IrDA mode.

Macro `FEATURE_USART_LIN_SLAVE`

```
#define FEATURE_USART_LIN_SLAVE
```

LIN slave mode.

Macro `FEATURE_USART_COLLISION_DECTION`

```
#define FEATURE_USART_COLLISION_DECTION
```

USART collision detection.

Macro FEATURE_USART_START_FRAME_DECTION

```
#define FEATURE_USART_START_FRAME_DECTION
```

USART start frame detection.

Macro FEATURE_USART_IMMEDIATE_BUFFER_OVERFLOW_NOTIFICATION

```
#define FEATURE_USART_IMMEDIATE_BUFFER_OVERFLOW_NOTIFICATION
```

USART start buffer overflow notification.

Macro FEATURE_USART_ISO7816

```
#define FEATURE_USART_ISO7816
```

ISO7816 for smart card interfacing.

Macro FEATURE_USART_LIN_MASTER

```
#define FEATURE_USART_LIN_MASTER
```

LIN master mode.

Macro FEATURE_USART_RS485

```
#define FEATURE_USART_RS485
```

RS485 mode.

16.6.3.2. Macro PINMUX_DEFAULT

```
#define PINMUX_DEFAULT
```

Default pinmux

16.6.3.3. Macro PINMUX_UNUSED

```
#define PINMUX_UNUSED
```

Unused pinmux

16.6.3.4. Macro USART_TIMEOUT

```
#define USART_TIMEOUT
```

USART timeout value

16.6.4. Function Definitions

16.6.4.1. Lock/Unlock

Function `usart_lock()`

Attempt to get lock on driver instance.

```
enum status_code usart_lock(  
    struct usart_module *const module)
```

This function checks the instance's lock, which indicates whether or not it is currently in use, and sets the lock if it was not already set.

The purpose of this is to enable exclusive access to driver instances, so that, e.g., transactions by different services will not interfere with each other.

Table 16-4. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 16-5. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

Function `usart_unlock()`

Unlock driver instance.

```
void usart_unlock(  
    struct usart_module *const module)
```

This function clears the instance lock, indicating that it is available for use.

Table 16-6. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

16.6.4.2. Writing and Reading

Function `usart_write_wait()`

Transmit a character via the USART.

```
enum status_code usart_write_wait(  
    struct usart_module *const module,  
    const uint16_t tx_data)
```

This blocking function will transmit a single character via the USART.

Table 16-7. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Data to transfer

Returns

Status of the operation.

Table 16-8. Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_BUSY	If the operation was not completed, due to the USART module being busy
STATUS_ERR_DENIED	If the transmitter is not enabled

Function usart_read_wait()

Receive a character via the USART.

```
enum status_code usart_read_wait(
    struct usart_module *const module,
    uint16_t *const rx_data)
```

This blocking function will receive a character via the USART.

Table 16-9. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[out]	rx_data	Pointer to received data

Returns

Status of the operation.

Table 16-10. Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_BUSY	If the operation was not completed, due to the USART module being busy
STATUS_ERR_BAD_FORMAT	If the operation was not completed, due to configuration mismatch between USART and the sender
STATUS_ERR_BAD_OVERFLOW	If the operation was not completed, due to the baudrate being too low or the system frequency being too high
STATUS_ERR_BAD_DATA	If the operation was not completed, due to data being corrupted
STATUS_ERR_DENIED	If the receiver is not enabled

Function usart_write_buffer_wait()

Transmit a buffer of characters via the USART.

```
enum status_code usart_write_buffer_wait(
    struct usart_module *const module,
    const uint8_t * tx_data,
    uint16_t length)
```

This blocking function will transmit a block of `length` characters via the USART.

Note: Using this function in combination with the interrupt (`_job`) functions is not recommended as it has no functionality to check if there is an ongoing interrupt driven operation running or not.

Table 16-11. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	tx_data	Pointer to data to transmit
[in]	length	Number of characters to transmit

Note: If using 9-bit data, the array that `*tx_data` point to should be defined as `uint16_t` array and should be casted to `uint8_t*` pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define TX_LEN 3
uint16_t tx_buf[TX_LEN] = {0x0111, 0x0022, 0x0133};
usart_write_buffer_wait(&module, (uint8_t*)tx_buf, TX_LEN);
```

Returns

Status of the operation.

Table 16-12. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID_ARG	If operation was not completed, due to invalid arguments
STATUS_ERR_TIMEOUT	If operation was not completed, due to USART module timing out
STATUS_ERR_DENIED	If the transmitter is not enabled

Function `usart_read_buffer_wait()`

Receive a buffer of `length` characters via the USART.

```
enum status_code usart_read_buffer_wait(
    struct usart_module *const module,
    uint8_t * rx_data,
    uint16_t length)
```

This blocking function will receive a block of `length` characters via the USART.

Note: Using this function in combination with the interrupt (`*_job`) functions is not recommended as it has no functionality to check if there is an ongoing interrupt driven operation running or not.

Table 16-13. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[out]	rx_data	Pointer to receive buffer
[in]	length	Number of characters to receive

Note: If using 9-bit data, the array that *rx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define RX_LEN 3
uint16_t rx_buf[RX_LEN] = {0x0,};
usart_read_buffer_wait(&module, (uint8_t*)rx_buf, RX_LEN);
```

Returns

Status of the operation.

Table 16-14. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID_ARG	If operation was not completed, due to an invalid argument being supplied
STATUS_ERR_TIMEOUT	If operation was not completed, due to USART module timing out
STATUS_ERR_BAD_FORMAT	If the operation was not completed, due to a configuration mismatch between USART and the sender
STATUS_ERR_BAD_OVERFLOW	If the operation was not completed, due to the baudrate being too low or the system frequency being too high
STATUS_ERR_BAD_DATA	If the operation was not completed, due to data being corrupted
STATUS_ERR_DENIED	If the receiver is not enabled

16.6.4.3. Enabling/Disabling Receiver and Transmitter

Function usart_enable_transceiver()

Enable Transceiver.

```
void usart_enable_transceiver(
    struct usart_module *const module,
    enum usart_transceiver_type transceiver_type)
```

Enable the given transceiver. Either RX or TX.

Table 16-15. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transceiver type

Function `usart_disable_transceiver()`

Disable Transceiver.

```
void usart_disable_transceiver(
    struct usart_module *const module,
    enum usart_transceiver_type transceiver_type)
```

Disable the given transceiver (RX or TX).

Table 16-16. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transceiver type

16.6.4.4. LIN Master Command and Status**Function `lin_master_send_cmd()`**

Sending LIN command.

```
void lin_master_send_cmd(
    struct usart_module *const module,
    enum lin_master_cmd cmd)
```

Sending LIN command.

Table 16-17. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	cmd	Cammand type

Function `lin_master_transmission_status()`

Get LIN transmission status.

```
bool lin_master_transmission_status(
    struct usart_module *const module)
```

Get LIN transmission status.

Table 16-18. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct

Returns

Status of LIN master transmission.

Table 16-19. Return Values

Return value	Description
true	Data transmission completed
false	Transmission is ongoing

16.6.4.5. Callback Management

Function `usart_register_callback()`

Registers a callback.

```
void usart_register_callback(  
    struct usart_module *const module,  
    usart_callback_t callback_func,  
    enum usart_callback callback_type)
```

Registers a callback function, which is implemented by the user.

Note: The callback must be enabled by `usart_enable_callback` in order for the interrupt handler to call it when the conditions for the callback type are met.

Table 16-20. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

Function `usart_unregister_callback()`

Unregisters a callback.

```
void usart_unregister_callback(  
    struct usart_module * module,  
    enum usart_callback callback_type)
```

Unregisters a callback function, which is implemented by the user.

Table 16-21. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to USART software instance struct
[in]	callback_type	Callback type given by an enum

Function `usart_enable_callback()`

Enables callback.

```
void usart_enable_callback(  
    struct usart_module *const module,  
    enum usart_callback callback_type)
```

Enables the callback function registered by the [usart_register_callback](#). The callback function will be called from the interrupt handler when the conditions for the callback type are met.

Table 16-22. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	callback_type	Callback type given by an enum

Function `usart_disable_callback()`

Disable callback.

```
void usart_disable_callback(  
    struct usart_module *const module,  
    enum usart_callback callback_type)
```

Disables the callback function registered by the [usart_register_callback](#), and the callback will not be called from the interrupt routine.

Table 16-23. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	callback_type	Callback type given by an enum

16.6.4.6. Writing and Reading

Function `usart_write_job()`

Asynchronous write a single char.

```
enum status_code usart_write_job(  
    struct usart_module *const module,  
    const uint16_t * tx_data)
```

Sets up the driver to write the data given. If registered and enabled, a callback function will be called when the transmit is completed.

Table 16-24. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	tx_data	Data to transfer

Returns

Status of the operation.

Table 16-25. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_BUSY	If operation was not completed, due to the USART module being busy
STATUS_ERR_DENIED	If the transmitter is not enabled

Function usart_read_job()

Asynchronous read a single char.

```
enum status_code usart_read_job(
    struct usart_module *const module,
    uint16_t *const rx_data)
```

Sets up the driver to read data from the USART module to the data pointer given. If registered and enabled, a callback will be called when the receiving is completed.

Table 16-26. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[out]	rx_data	Pointer to where received data should be put

Returns

Status of the operation.

Table 16-27. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_BUSY	If operation was not completed

Function usart_write_buffer_job()

Asynchronous buffer write.

```
enum status_code usart_write_buffer_job(
    struct usart_module *const module,
    uint8_t * tx_data,
    uint16_t length)
```

Sets up the driver to write a given buffer over the USART. If registered and enabled, a callback function will be called.

Table 16-28. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	tx_data	Pointer do data buffer to transmit
[in]	length	Length of the data to transmit

Note: If using 9-bit data, the array that *tx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define TX_LEN 3
uint16_t tx_buf[TX_LEN] = {0x0111, 0x0022, 0x0133};
usart_write_buffer_job(&module, (uint8_t*)tx_buf, TX_LEN);
```

Returns

Status of the operation.

Table 16-29. Return Values

Return value	Description
STATUS_OK	If operation was completed successfully.
STATUS_BUSY	If operation was not completed, due to the USART module being busy
STATUS_ERR_INVALID_ARG	If operation was not completed, due to invalid arguments
STATUS_ERR_DENIED	If the transmitter is not enabled

Function usart_read_buffer_job()

Asynchronous buffer read.

```
enum status_code usart_read_buffer_job(
    struct usart_module *const module,
    uint8_t * rx_data,
    uint16_t length)
```

Sets up the driver to read from the USART to a given buffer. If registered and enabled, a callback function will be called.

Table 16-30. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[out]	rx_data	Pointer to data buffer to receive
[in]	length	Data buffer length

Note: If using 9-bit data, the array that *rx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define RX_LEN 3
uint16_t rx_buf[RX_LEN] = {0x0,};
usart_read_buffer_job(&module, (uint8_t*)rx_buf, RX_LEN);
```

Returns

Status of the operation.

Table 16-31. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_BUSY	If operation was not completed, due to the USART module being busy
STATUS_ERR_INVALID_ARG	If operation was not completed, due to invalid arguments
STATUS_ERR_DENIED	If the transmitter is not enabled

Function usart_abort_job()

Cancels ongoing read/write operation.

```
void usart_abort_job(
    struct usart_module *const module,
    enum usart_transceiver_type transceiver_type)
```

Cancels the ongoing read/write operation modifying parameters in the USART software struct.

Table 16-32. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transfer type to cancel

Function usart_get_job_status()

Get status from the ongoing or last asynchronous transfer operation.

```
enum status_code usart_get_job_status(
    struct usart_module *const module,
    enum usart_transceiver_type transceiver_type)
```

Returns the error from a given ongoing or last asynchronous transfer operation. Either from a read or write transfer.

Table 16-33. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transfer type to check

Returns

Status of the given job.

Table 16-34. Return Values

Return value	Description
STATUS_OK	No error occurred during the last transfer
STATUS_BUSY	A transfer is ongoing
STATUS_ERR_BAD_DATA	The last operation was aborted due to a parity error. The transfer could be affected by external noise
STATUS_ERR_BAD_FORMAT	The last operation was aborted due to a frame error
STATUS_ERR_OVERFLOW	The last operation was aborted due to a buffer overflow
STATUS_ERR_INVALID_ARG	An invalid transceiver enum given

16.6.4.7. Function `usart_disable()`

Disable module.

```
void usart_disable(  
    const struct usart_module *const module)
```

Disables the USART module.

Table 16-35. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct

16.6.4.8. Function `usart_enable()`

Enable the module.

```
void usart_enable(  
    const struct usart_module *const module)
```

Enables the USART module.

Table 16-36. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct

16.6.4.9. Function `usart_get_config_defaults()`

Initializes the device to predefined defaults.

```
void usart_get_config_defaults(  
    struct usart_config *const config)
```

Initialize the USART device to predefined defaults:

- 8-bit asynchronous USART

- No parity
- One stop bit
- 9600 baud
- Transmitter enabled
- Receiver enabled
- GCLK generator 0 as clock source
- Default pin configuration

The configuration struct will be updated with the default configuration.

Table 16-37. Parameters

Data direction	Parameter name	Description
[in, out]	config	Pointer to configuration struct

16.6.4.10. Function `usart_init()`

Initializes the device.

```
enum status_code usart_init(
    struct usart_module *const module,
    Sercom *const hw,
    const struct usart_config *const config)
```

Initializes the USART device based on the setting specified in the configuration struct.

Table 16-38. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to USART device
[in]	hw	Pointer to USART hardware instance
[in]	config	Pointer to configuration struct

Returns

Status of the initialization.

Table 16-39. Return Values

Return value	Description
STATUS_OK	The initialization was successful
STATUS_BUSY	The USART module is busy resetting
STATUS_ERR_DENIED	The USART has not been disabled in advance of initialization
STATUS_ERR_INVALID_ARG	The configuration struct contains invalid configuration
STATUS_ERR_ALREADY_INITIALIZED	The SERCOM instance has already been initialized with different clock configuration
STATUS_ERR_BAUD_UNAVAILABLE	The BAUD rate given by the configuration struct cannot be reached with the current clock configuration

16.6.4.11. Function `usart_is_syncing()`

Check if peripheral is busy syncing registers across clock domains.

```
bool usart_is_syncing(  
    const struct usart_module *const module)
```

Return peripheral synchronization status. If doing a non-blocking implementation this function can be used to check the sync state and hold off any new actions until sync is complete. If this function is not run; the functions will block until the sync has completed.

Table 16-40. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to peripheral module

Returns

Peripheral sync status.

Table 16-41. Return Values

Return value	Description
true	Peripheral is busy syncing
false	Peripheral is not busy syncing and can be read/written without stalling the bus

16.6.4.12. Function `usart_reset()`

Resets the USART module.

```
void usart_reset(  
    const struct usart_module *const module)
```

Disables and resets the USART module.

Table 16-42. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the USART software instance struct

16.6.5. Enumeration Definitions

16.6.5.1. Enum `iso7816_guard_time`

The value of ISO7816 guard time.

Table 16-43. Members

Enum value	Description
ISO7816_GUARD_TIME_2_BIT	The guard time is 2-bit times
ISO7816_GUARD_TIME_3_BIT	The guard time is 3-bit times
ISO7816_GUARD_TIME_4_BIT	The guard time is 4-bit times

Enum value	Description
ISO7816_GUARD_TIME_5_BIT	The guard time is 5-bit times
ISO7816_GUARD_TIME_6_BIT	The guard time is 6-bit times
ISO7816_GUARD_TIME_7_BIT	The guard time is 7-bit times

16.6.5.2. Enum iso7816_inhibit_nack

The value of ISO7816 receive NACK inhibit.

Table 16-44. Members

Enum value	Description
ISO7816_INHIBIT_NACK_DISABLE	The NACK is generated
ISO7816_INHIBIT_NACK_ENABLE	The NACK is not generated

16.6.5.3. Enum iso7816_protocol_type

ISO7816 protocol type.

Table 16-45. Members

Enum value	Description
ISO7816_PROTOCOL_T_0	ISO7816 protocol type 0
ISO7816_PROTOCOL_T_1	ISO7816 protocol type 1

16.6.5.4. Enum iso7816_successive_recv_nack

The value of ISO7816 disable successive receive NACK.

Table 16-46. Members

Enum value	Description
ISO7816_SUCCESSIVE_RECV_NACK_DISABLE	The successive receive NACK is enable.
ISO7816_SUCCESSIVE_RECV_NACK_ENABLE	The successive receive NACK is disable.

16.6.5.5. Enum lin_master_break_length

Length of the break field transmitted when in LIN master mode

Table 16-47. Members

Enum value	Description
LIN_MASTER_BREAK_LENGTH_13_BIT	Break field transmission is 13 bit times
LIN_MASTER_BREAK_LENGTH_17_BIT	Break field transmission is 17 bit times
LIN_MASTER_BREAK_LENGTH_21_BIT	Break field transmission is 21 bit times
LIN_MASTER_BREAK_LENGTH_26_BIT	Break field transmission is 26 bit times

16.6.5.6. Enum lin_master_cmd

LIN master command enum.

Table 16-48. Members

Enum value	Description
LIN_MASTER_SOFTWARE_CONTROL_TRANSMIT_CMD	LIN master software control transmission command
LIN_MASTER_AUTO_TRANSMIT_CMD	LIN master automatically transmission command

16.6.5.7. Enum lin_master_header_delay

LIN master header delay between break and sync transmission, and between the sync and identifier (ID) fields. This field is only valid when using automatically transmission command

Table 16-49. Members

Enum value	Description
LIN_MASTER_HEADER_DELAY_0	Delay between break and sync transmission is 1 bit time. Delay between sync and ID transmission is 1 bit time.
LIN_MASTER_HEADER_DELAY_1	Delay between break and sync transmission is 4 bit time. Delay between sync and ID transmission is 4 bit time.
LIN_MASTER_HEADER_DELAY_2	Delay between break and sync transmission is 8 bit time. Delay between sync and ID transmission is 4 bit time.
LIN_MASTER_HEADER_DELAY_3	Delay between break and sync transmission is 14 bit time. Delay between sync and ID transmission is 4 bit time.

16.6.5.8. Enum lin_node_type

LIN node type.

Table 16-50. Members

Enum value	Description
LIN_MASTER_NODE	LIN master mode
LIN_SLAVE_NODE	LIN slave mode
LIN_INVALID_MODE	Neither LIN master nor LIN slave mode

16.6.5.9. Enum rs485_guard_time

The value of RS485 guard time.

Table 16-51. Members

Enum value	Description
RS485_GUARD_TIME_0_BIT	The guard time is 0-bit time
RS485_GUARD_TIME_1_BIT	The guard time is 1-bit time
RS485_GUARD_TIME_2_BIT	The guard time is 2-bit times
RS485_GUARD_TIME_3_BIT	The guard time is 3-bit times
RS485_GUARD_TIME_4_BIT	The guard time is 4-bit times
RS485_GUARD_TIME_5_BIT	The guard time is 5-bit times
RS485_GUARD_TIME_6_BIT	The guard time is 6-bit times
RS485_GUARD_TIME_7_BIT	The guard time is 7-bit times

16.6.5.10. Enum usart_callback

Callbacks for the Asynchronous USART driver.

Table 16-52. Members

Enum value	Description
USART_CALLBACK_BUFFER_TRANSMITTED	Callback for buffer transmitted
USART_CALLBACK_BUFFER_RECEIVED	Callback for buffer received
USART_CALLBACK_ERROR	Callback for error
USART_CALLBACK_BREAK_RECEIVED	Callback for break character is received
USART_CALLBACK_CTS_INPUT_CHANGE	Callback for a change is detected on the CTS pin
USART_CALLBACK_START_RECEIVED	Callback for a start condition is detected on the RxD line

16.6.5.11. Enum usart_character_size

Number of bits for the character sent in a frame.

Table 16-53. Members

Enum value	Description
USART_CHARACTER_SIZE_5BIT	The char being sent in a frame is five bits long
USART_CHARACTER_SIZE_6BIT	The char being sent in a frame is six bits long
USART_CHARACTER_SIZE_7BIT	The char being sent in a frame is seven bits long
USART_CHARACTER_SIZE_8BIT	The char being sent in a frame is eight bits long
USART_CHARACTER_SIZE_9BIT	The char being sent in a frame is nine bits long

16.6.5.12. Enum usart_dataorder

The data order decides which MSB or LSB is shifted out first when data is transferred.

Table 16-54. Members

Enum value	Description
USART_DATAORDER_MSB	The MSB will be shifted out first during transmission, and shifted in first during reception
USART_DATAORDER_LSB	The LSB will be shifted out first during transmission, and shifted in first during reception

16.6.5.13. Enum usart_parity

Select parity USART parity mode.

Table 16-55. Members

Enum value	Description
USART_PARITY_ODD	For odd parity checking, the parity bit will be set if number of ones being transferred is even
USART_PARITY_EVEN	For even parity checking, the parity bit will be set if number of ones being received is odd
USART_PARITY_NONE	No parity checking will be executed, and there will be no parity bit in the received frame

16.6.5.14. Enum usart_sample_adjustment

The value of sample number used for majority voting.

Table 16-56. Members

Enum value	Description
USART_SAMPLE_ADJUSTMENT_7_8_9	The first, middle and last sample number used for majority voting is 7-8-9
USART_SAMPLE_ADJUSTMENT_9_10_11	The first, middle and last sample number used for majority voting is 9-10-11
USART_SAMPLE_ADJUSTMENT_11_12_13	The first, middle and last sample number used for majority voting is 11-12-13
USART_SAMPLE_ADJUSTMENT_13_14_15	The first, middle and last sample number used for majority voting is 13-14-15

16.6.5.15. Enum usart_sample_rate

The value of sample rate and baudrate generation mode.

Table 16-57. Members

Enum value	Description
USART_SAMPLE_RATE_16X_ARITHMETIC	16x over-sampling using arithmetic baudrate generation
USART_SAMPLE_RATE_16X_FRACTIONAL	16x over-sampling using fractional baudrate generation
USART_SAMPLE_RATE_8X_ARITHMETIC	8x over-sampling using arithmetic baudrate generation
USART_SAMPLE_RATE_8X_FRACTIONAL	8x over-sampling using fractional baudrate generation
USART_SAMPLE_RATE_3X_ARITHMETIC	3x over-sampling using arithmetic baudrate generation

16.6.5.16. Enum `usart_signal_mux_settings`

Set the functionality of the SERCOM pins.

See [SERCOM USART MUX Settings](#) for a description of the various MUX setting options.

Table 16-58. Members

Enum value	Description
USART_RX_0_TX_0_XCK_1	MUX setting RX_0_TX_0_XCK_1
USART_RX_0_TX_2_XCK_3	MUX setting RX_0_TX_2_XCK_3
USART_RX_0_TX_0_RTS_2_CTS_3	MUX setting USART_RX_0_TX_0_RTS_2_CTS_3
USART_RX_1_TX_0_XCK_1	MUX setting RX_1_TX_0_XCK_1
USART_RX_1_TX_2_XCK_3	MUX setting RX_1_TX_2_XCK_3
USART_RX_1_TX_0_RTS_2_CTS_3	MUX setting USART_RX_1_TX_0_RTS_2_CTS_3
USART_RX_2_TX_0_XCK_1	MUX setting RX_2_TX_0_XCK_1
USART_RX_2_TX_2_XCK_3	MUX setting RX_2_TX_2_XCK_3
USART_RX_2_TX_0_RTS_2_CTS_3	MUX setting USART_RX_2_TX_0_RTS_2_CTS_3
USART_RX_3_TX_0_XCK_1	MUX setting RX_3_TX_0_XCK_1
USART_RX_3_TX_2_XCK_3	MUX setting RX_3_TX_2_XCK_3
USART_RX_3_TX_0_RTS_2_CTS_3	MUX setting USART_RX_3_TX_0_RTS_2_CTS_3
USART_RX_0_TX_0_XCK_1_TE_2	MUX setting USART_RX_0_TX_0_XCK_1_TE_2
USART_RX_1_TX_0_XCK_1_TE_2	MUX setting USART_RX_1_TX_0_XCK_1_TE_2
USART_RX_2_TX_0_XCK_1_TE_2	MUX setting USART_RX_2_TX_0_XCK_1_TE_2
USART_RX_3_TX_0_XCK_1_TE_2	MUX setting USART_RX_3_TX_0_XCK_1_TE_2

16.6.5.17. Enum `usart_stopbits`

Number of stop bits for a frame.

Table 16-59. Members

Enum value	Description
USART_STOPBITS_1	Each transferred frame contains one stop bit
USART_STOPBITS_2	Each transferred frame contains two stop bits

16.6.5.18. Enum usart_transceiver_type

Select Receiver or Transmitter.

Table 16-60. Members

Enum value	Description
USART_TRANSCEIVER_RX	The parameter is for the Receiver
USART_TRANSCEIVER_TX	The parameter is for the Transmitter

16.6.5.19. Enum usart_transfer_mode

Select USART transfer mode.

Table 16-61. Members

Enum value	Description
USART_TRANSFER_SYNCHRONOUSLY	Transfer of data is done synchronously
USART_TRANSFER_ASYNCHRONOUSLY	Transfer of data is done asynchronously

16.7. Extra Information for SERCOM USART Driver**16.7.1. Acronyms**

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
SERCOM	Serial Communication Interface
USART	Universal Synchronous and Asynchronous Serial Receiver and Transmitter
LSB	Least Significant Bit
MSB	Most Significant Bit
DMA	Direct Memory Access

16.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver
- System clock configuration

16.7.3. Errata

There are no errata related to this driver.

16.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added new feature as below: • ISO7816
Added new features as below: • LIN master • RS485
Added new features as below: • Oversample • Buffer overflow notification • Irda • Lin slave • Start frame detection • Hardware flow control • Collision detection • DMA support
• Added new <code>transmitter_enable</code> and <code>receiver_enable</code> Boolean values to struct <code>usart_config</code> • Altered <code>usart_write_*</code> and <code>usart_read_*</code> functions to abort with an error code if the relevant transceiver is not enabled • Fixed <code>usart_write_buffer_wait()</code> and <code>usart_read_buffer_wait()</code> not aborting correctly when a timeout condition occurs
Initial Release

16.8. Examples for SERCOM USART Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Serial USART \(SERCOM USART\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for SERCOM USART - Basic](#)
- [Quick Start Guide for SERCOM USART - Callback](#)
- [Quick Start Guide for Using DMA with SERCOM USART](#)
- [Quick Start Guide for SERCOM USART LIN](#)

16.8.1. Quick Start Guide for SERCOM USART - Basic

This quick start will echo back characters typed into the terminal. In this use case the USART will be configured with the following settings:

- Asynchronous mode
- 9600 Baudrate
- 8-bits, No Parity and one Stop Bit
- TX and RX enabled and connected to the Xplained Pro Embedded Debugger virtual COM port

16.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, outside of any functions:

```
struct usart_module usart_instance;
```

Copy-paste the following setup code to your user application:

```
void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);

    config_usart.baudrate      = 9600;
    config_usart.mux_setting   = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0   = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1   = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2   = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3   = EDBG_CDC_SERCOM_PINMUX_PAD3;

    while (usart_init(&usart_instance,
                     EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
    }

    usart_enable(&usart_instance);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_usart();
```

Workflow

1. Create a module software instance structure for the USART module to store the USART driver state while it is in use.

```
struct usart_module usart_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the USART module.
 1. Create a USART module configuration struct, which can be filled out to adjust the configuration of a physical USART peripheral.

```
struct usart_config config_usart;
```


2. Initialize the USART configuration struct with the module's default values.

```
usart_get_config_defaults(&config_usart);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the USART settings to configure the physical pinout, baudrate, and other relevant parameters.

```
config_usart.baudrate      = 9600;
config_usart.mux_setting   = EDBG_CDC_SERCOM_MUX_SETTING;
config_usart.pinmux_pad0   = EDBG_CDC_SERCOM_PINMUX_PAD0;
config_usart.pinmux_pad1   = EDBG_CDC_SERCOM_PINMUX_PAD1;
config_usart.pinmux_pad2   = EDBG_CDC_SERCOM_PINMUX_PAD2;
config_usart.pinmux_pad3   = EDBG_CDC_SERCOM_PINMUX_PAD3;
```

4. Configure the USART module with the desired settings, retrying while the driver is busy until the configuration is stressfully set.

```
while (usart_init(&usart_instance,
                 EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
}
```

5. Enable the USART module.

```
usart_enable(&usart_instance);
```

16.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
uint8_t string[] = "Hello World!\r\n";
usart_write_buffer_wait(&usart_instance, string, sizeof(string));

uint16_t temp;

while (true) {
    if (usart_read_wait(&usart_instance, &temp) == STATUS_OK) {
        while (usart_write_wait(&usart_instance, temp) != STATUS_OK) {
        }
    }
}
```

Workflow

1. Send a string to the USART to show the demo is running, blocking until all characters have been sent.

```
uint8_t string[] = "Hello World!\r\n";
usart_write_buffer_wait(&usart_instance, string, sizeof(string));
```

2. Enter an infinite loop to continuously echo received values on the USART.

```
while (true) {
    if (usart_read_wait(&usart_instance, &temp) == STATUS_OK) {
        while (usart_write_wait(&usart_instance, temp) != STATUS_OK) {
        }
    }
}
```

3. Perform a blocking read of the USART, storing the received character into the previously declared temporary variable.

```
if (usart_read_wait(&usart_instance, &temp) == STATUS_OK) {
```

4. Echo the received variable back to the USART via a blocking write.

```
while (usart_write_wait(&usart_instance, temp) != STATUS_OK) {  
}
```

16.8.2. Quick Start Guide for SERCOM USART - Callback

This quick start will echo back characters typed into the terminal, using asynchronous TX and RX callbacks from the USART peripheral. In this use case the USART will be configured with the following settings:

- Asynchronous mode
- 9600 Baudrate
- 8-bits, No Parity and one Stop Bit
- TX and RX enabled and connected to the Xplained Pro Embedded Debugger virtual COM port

16.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, outside of any functions:

```
struct usart_module usart_instance;
```

```
#define MAX_RX_BUFFER_LENGTH 5
```

```
volatile uint8_t rx_buffer[MAX_RX_BUFFER_LENGTH];
```

Copy-paste the following callback function code to your user application:

```
void usart_read_callback(struct usart_module *const usart_module)  
{  
    usart_write_buffer_job(&usart_instance,  
        (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);  
}  
  
void usart_write_callback(struct usart_module *const usart_module)  
{  
    port_pin_toggle_output_level(LED_0_PIN);  
}
```

Copy-paste the following setup code to your user application:

```
void configure_usart(void)  
{  
    struct usart_config config_usart;  
    usart_get_config_defaults(&config_usart);  
  
    config_usart.baudrate = 9600;  
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;  
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;  
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;  
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;  
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
```

```

while (usart_init(&usart_instance,
    EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
}

usart_enable(&usart_instance);
}

void configure_usart_callbacks(void)
{
    usart_register_callback(&usart_instance,
        usart_write_callback, USART_CALLBACK_BUFFER_TRANSMITTED);
    usart_register_callback(&usart_instance,
        usart_read_callback, USART_CALLBACK_BUFFER_RECEIVED);

    usart_enable_callback(&usart_instance,
        USART_CALLBACK_BUFFER_TRANSMITTED);
    usart_enable_callback(&usart_instance, USART_CALLBACK_BUFFER_RECEIVED);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_usart();
configure_usart_callbacks();

```

Workflow

1. Create a module software instance structure for the USART module to store the USART driver state while it is in use.

```
struct usart_module usart_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the USART module.
 1. Create a USART module configuration struct, which can be filled out to adjust the configuration of a physical USART peripheral.

```
struct usart_config config_usart;
```

2. Initialize the USART configuration struct with the module's default values.

```
usart_get_config_defaults(&config_usart);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the USART settings to configure the physical pinout, baudrate, and other relevant parameters.

```

config_usart.baudrate      = 9600;
config_usart.mux_setting  = EDBG_CDC_SERCOM_MUX_SETTING;
config_usart.pinmux_pad0   = EDBG_CDC_SERCOM_PINMUX_PAD0;
config_usart.pinmux_pad1   = EDBG_CDC_SERCOM_PINMUX_PAD1;
config_usart.pinmux_pad2   = EDBG_CDC_SERCOM_PINMUX_PAD2;
config_usart.pinmux_pad3   = EDBG_CDC_SERCOM_PINMUX_PAD3;

```

4. Configure the USART module with the desired settings, retrying while the driver is busy until the configuration is stressfully set.

```

while (usart_init(&usart_instance,
    EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
}

```

5. Enable the USART module.

```
usart_enable(&usart_instance);
```

3. Configure the USART callbacks.

1. Register the TX and RX callback functions with the driver.

```
usart_register_callback(&usart_instance,  
    usart_write_callback, USART_CALLBACK_BUFFER_TRANSMITTED);  
usart_register_callback(&usart_instance,  
    usart_read_callback, USART_CALLBACK_BUFFER_RECEIVED);
```

2. Enable the TX and RX callbacks so that they will be called by the driver when appropriate.

```
usart_enable_callback(&usart_instance,  
    USART_CALLBACK_BUFFER_TRANSMITTED);  
usart_enable_callback(&usart_instance,  
    USART_CALLBACK_BUFFER_RECEIVED);
```

16.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();  
  
uint8_t string[] = "Hello World!\r\n";  
usart_write_buffer_wait(&usart_instance, string, sizeof(string));  
  
while (true) {  
    usart_read_buffer_job(&usart_instance,  
        (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);  
}
```

Workflow

1. Enable global interrupts, so that the callbacks can be fired.

```
system_interrupt_enable_global();
```

2. Send a string to the USART to show the demo is running, blocking until all characters have been sent.

```
uint8_t string[] = "Hello World!\r\n";  
usart_write_buffer_wait(&usart_instance, string, sizeof(string));
```

3. Enter an infinite loop to continuously echo received values on the USART.

```
while (true) {
```

4. Perform an asynchronous read of the USART, which will fire the registered callback when characters are received.

```
usart_read_buffer_job(&usart_instance,  
    (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);
```

16.8.3. Quick Start Guide for Using DMA with SERCOM USART

The supported board list:

- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM D11 Xplained Pro

- SAM DA1 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM C21 Xplained Pro

This quick start will receive eight bytes of data from the PC terminal and transmit back the string to the terminal through DMA. In this use case the USART will be configured with the following settings:

- Asynchronous mode
- 9600 Baudrate
- 8-bits, No Parity and one Stop Bit
- TX and RX enabled and connected to the Xplained Pro Embedded Debugger virtual COM port

16.8.3.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Add to the main application source file, outside of any functions:

```
struct usart_module usart_instance;
```

```
struct dma_resource usart_dma_resource_rx;
struct dma_resource usart_dma_resource_tx;
```

```
#define BUFFER_LEN 8
static uint16_t string[BUFFER_LEN];
```

```
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_rx SECTION_DMAC_DESCRIPTOR;
DmacDescriptor example_descriptor_tx SECTION_DMAC_DESCRIPTOR;
```

Copy-paste the following setup code to your user application:

```
static void transfer_done_rx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_tx);
}

static void transfer_done_tx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_rx);
}

static void configure_dma_resource_rx(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_RX;
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(resource, &config);
}

static void setup_transfer_descriptor_rx(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;
```

```

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.src_increment_enable = false;
    descriptor_config.block_transfer_count = BUFFER_LEN;
    descriptor_config.destination_address =
        (uint32_t)string + sizeof(string);
    descriptor_config.source_address =
        (uint32_t)(&usart_instance.hw->USART.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}

static void configure_dma_resource_tx(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_TX;
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(resource, &config);
}

static void setup_transfer_descriptor_tx(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.block_transfer_count = BUFFER_LEN;
    descriptor_config.source_address = (uint32_t)string + sizeof(string);
    descriptor_config.destination_address =
        (uint32_t)(&usart_instance.hw->USART.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}

static void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);

    config_usart.baudrate = 9600;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;

    while (usart_init(&usart_instance,
        EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
    }

    usart_enable(&usart_instance);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_usart();

configure_dma_resource_rx(&usart_dma_resource_rx);

```

```

configure_dma_resource_tx(&usart_dma_resource_tx);

setup_transfer_descriptor_rx(&example_descriptor_rx);
setup_transfer_descriptor_tx(&example_descriptor_tx);

dma_add_descriptor(&usart_dma_resource_rx, &example_descriptor_rx);
dma_add_descriptor(&usart_dma_resource_tx, &example_descriptor_tx);

dma_register_callback(&usart_dma_resource_rx, transfer_done_rx,
    DMA_CALLBACK_TRANSFER_DONE);
dma_register_callback(&usart_dma_resource_tx, transfer_done_tx,
    DMA_CALLBACK_TRANSFER_DONE);

dma_enable_callback(&usart_dma_resource_rx,
    DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&usart_dma_resource_tx,
    DMA_CALLBACK_TRANSFER_DONE);

```

Workflow

Create variables

1. Create a module software instance structure for the USART module to store the USART driver state while it is in use.

```
struct usart_module usart_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create module software instance structures for DMA resources to store the DMA resource state while it is in use.

```
struct dma_resource usart_dma_resource_rx;
struct dma_resource usart_dma_resource_tx;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

3. Create a buffer to store the data to be transferred /received.

```
#define BUFFER_LEN 8
static uint16_t string[BUFFER_LEN];
```

4. Create DMA transfer descriptors for RX/TX.

```
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_rx SECTION_DMAC_DESCRIPTOR;
DmacDescriptor example_descriptor_tx SECTION_DMAC_DESCRIPTOR;
```

Configure the USART

1. Create a USART module configuration struct, which can be filled out to adjust the configuration of a physical USART peripheral.

```
struct usart_config config_usart;
```

2. Initialize the USART configuration struct with the module's default values.

```
usart_get_config_defaults(&config_usart);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the USART settings to configure the physical pinout, baudrate, and other relevant parameters.

```
config_usart.baudrate      = 9600;
config_usart.mux_setting  = EDBG_CDC_SERCOM_MUX_SETTING;
config_usart.pinmux_pad0  = EDBG_CDC_SERCOM_PINMUX_PAD0;
config_usart.pinmux_pad1  = EDBG_CDC_SERCOM_PINMUX_PAD1;
config_usart.pinmux_pad2  = EDBG_CDC_SERCOM_PINMUX_PAD2;
config_usart.pinmux_pad3  = EDBG_CDC_SERCOM_PINMUX_PAD3;
```

4. Configure the USART module with the desired settings, retrying while the driver is busy until the configuration is stressfully set.

```
while (usart_init(&usart_instance,
                  EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
}
```

5. Enable the USART module.

```
usart_enable(&usart_instance);
```

Configure DMA

1. Create a callback function of receiver done.

```
static void transfer_done_rx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_tx);
}
```

2. Create a callback function of transmission done.

```
static void transfer_done_tx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_rx);
}
```

3. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

4. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

5. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM TX empty trigger causes a beat transfer in this example.

```
config.peripheral_trigger = EDBG_CDC_SERCOM_DMACH_ID_RX;
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

6. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

7. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

8. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```


Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

9. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.src_increment_enable = false;
descriptor_config.block_transfer_count = BUFFER_LEN;
descriptor_config.destination_address =
    (uint32_t)string + sizeof(string);
descriptor_config.source_address =
    (uint32_t)(&usart_instance.hw->USART.DATA.reg);
```

10. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

11. Create a DMA resource configuration structure for TX, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

12. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

13. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM RX Ready trigger causes a beat transfer in this example.

```
config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_TX;
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

14. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

15. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

16. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

17. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.block_transfer_count = BUFFER_LEN;
descriptor_config.source_address = (uint32_t)string + sizeof(string);
descriptor_config.destination_address =
    (uint32_t)(&usart_instance.hw->USART.DATA.reg);
```

18. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

16.8.3.2. Use Case

Code

Copy-paste the following code to your user application:

```
dma_start_transfer_job(&usart_dma_resource_rx);  
  
while (true) {  
}
```

Workflow

1. Wait for receiving data.

```
dma_start_transfer_job(&usart_dma_resource_rx);
```

2. Enter endless loop.

```
while (true) {  
}
```

16.8.4. Quick Start Guide for SERCOM USART LIN

The supported board list:

- SAMC21 Xplained Pro

This quick start will set up LIN frame format transmission according to your configuration `CONF_LIN_NODE_TYPE`. For LIN master, it will send LIN command after startup. For LIN slave, once received a format from LIN master with ID `LIN_ID_FIELD_VALUE`, it will reply four data bytes plus a checksum.

16.8.4.1. Setup

Prerequisites

When verify data transmission between LIN master and slave, two boards are needed: one is for LIN master and the other is for LIN slave. connect LIN master LIN PIN with LIN slave LIN PIN.

Code

Add to the main application source file, outside of any functions:

```
static struct usart_module cdc_instance, lin_instance;  
  
#define LIN_ID_FIELD_VALUE 0x64  
  
#define LIN_DATA_LEN 5  
static uint8_t rx_buffer[LIN_DATA_LEN]={0};  
const static uint8_t tx_buffer[LIN_DATA_LEN]={0x4a, 0x55, 0x93, 0xe5, 0xe6};
```

Copy-paste the following setup code to your user application:

```
static void configure_usart_cdc(void)  
{  
  
    struct usart_config config_cdc;  
    usart_get_config_defaults(&config_cdc);  
    config_cdc.baudrate = 115200;  
    config_cdc.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;  
    config_cdc.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;  
    config_cdc.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;  
    config_cdc.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
```

```

    config_cdc.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&cdc_instance, EDBG_CDC_MODULE, &config_cdc);
    usart_enable(&cdc_instance);
}

static void lin_read_callback(struct usart_module *const usart_module)
{
    uint8_t i = 0;

    if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
        for(i = 0; i < LIN_DATA_LEN; i++){
            if(rx_buffer[i] != tx_buffer[i]) {
                printf("Data error\r\n");
                break;
            }
        }
        if(i == LIN_DATA_LEN){
            printf("Slave response: OK\r\n");
        }
    } else if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
        if(rx_buffer[0] == LIN_ID_FIELD_VALUE) {
            usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_TX);
            printf("Receive ID field from mater: OK \r\n");
            usart_write_buffer_job(&lin_instance,
                (uint8_t *)tx_buffer, LIN_DATA_LEN);
        }
    }
}

static void lin_read_error_callback(struct usart_module *const usart_module)
{
    printf("Data Read error\r\n");
}

static void configure_usart_lin(void)
{
    struct port_config pin_conf;
    port_get_config_defaults(&pin_conf);
    pin_conf.direction = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LIN_EN_PIN, &pin_conf);

    /* Enable LIN module*/
    port_pin_set_output_level(LIN_EN_PIN, 1);

    struct usart_config config_lin;
    usart_get_config_defaults(&config_lin);

    /* LIN frame format*/
    config_lin.lin_node = CONF_LIN_NODE_TYPE;
    config_lin.transfer_mode = USART_TRANSFER_ASYNCHRONOUSLY;
    config_lin.sample_rate = USART_SAMPLE_RATE_16X_FRACTIONAL;

    config_lin.baudrate = 115200;
    config_lin.mux_setting = LIN_USART_SERCOM_MUX_SETTING;
    config_lin.pinmux_pad0 = LIN_USART_SERCOM_PINMUX_PAD0;
    config_lin.pinmux_pad1 = LIN_USART_SERCOM_PINMUX_PAD1;
    config_lin.pinmux_pad2 = LIN_USART_SERCOM_PINMUX_PAD2;
    config_lin.pinmux_pad3 = LIN_USART_SERCOM_PINMUX_PAD3;

    /* Disable receiver and transmitter */
    config_lin.receiver_enable = false;
    config_lin.transmitter_enable = false;
}

```

```

if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
    config_lin.lin_slave_enable = true;
}

while (usart_init(&lin_instance,
    LIN_USART_MODULE, &config_lin) != STATUS_OK) {
}

usart_enable(&lin_instance);

usart_register_callback(&lin_instance,
    lin_read_callback, USART_CALLBACK_BUFFER_RECEIVED);
usart_enable_callback(&lin_instance, USART_CALLBACK_BUFFER_RECEIVED);
usart_register_callback(&lin_instance,
    lin_read_error_callback, USART_CALLBACK_ERROR);
usart_enable_callback(&lin_instance, USART_CALLBACK_ERROR);
system_interrupt_enable_global();
}

```

Add to user application initialization (typically the start of `main()`):

```

system_init();
configure_usart_cdc();

```

Workflow

1. Create USART CDC and LIN module software instance structure for the USART module to store the USART driver state while it is in use.

```
static struct usart_module cdc_instance, lin_instance;
```

2. Define LIN ID field for header format.

```
#define LIN_ID_FIELD_VALUE 0x64
```

Note: The ID `LIN_ID_FIELD_VALUE` is eight bits as `[P1,P0,ID5...ID0]`, when it's `0x64`, the data field length is four bytes plus a checksum byte.

3. Define LIN RX/TX buffer.

```

#define LIN_DATA_LEN 5
static uint8_t rx_buffer[LIN_DATA_LEN]={0};
const static uint8_t tx_buffer[LIN_DATA_LEN]={0x4a,0x55,0x93,0xe5,0xe6};

```

Note: For `tx_buffer` and `rx_buffer`, the last byte is for checksum.

4. Configure the USART CDC for output message.

```

static void configure_usart_cdc(void)
{
    struct usart_config config_cdc;
    usart_get_config_defaults(&config_cdc);
    config_cdc.baudrate = 115200;
    config_cdc.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_cdc.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_cdc.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_cdc.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_cdc.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&cdc_instance, EDBG_CDC_MODULE, &config_cdc);
    usart_enable(&cdc_instance);
}

```

5. Configure the USART LIN module.

```
static void lin_read_callback(struct usart_module *const usart_module)
{
    uint8_t i = 0;

    if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
        for(i = 0; i < LIN_DATA_LEN; i++){
            if(rx_buffer[i] != tx_buffer[i]) {
                printf("Data error\r\n");
                break;
            }
        }
        if(i == LIN_DATA_LEN){
            printf("Slave response: OK\r\n");
        }
    } else if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
        if(rx_buffer[0] == LIN_ID_FIELD_VALUE) {
            usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_TX);
            printf("Receive ID field from mater: OK \r\n");
            usart_write_buffer_job(&lin_instance,
                (uint8_t *)tx_buffer, LIN_DATA_LEN);
        }
    }
}

static void lin_read_error_callback(struct usart_module *const
usart_module)
{
    printf("Data Read error\r\n");
}

static void configure_usart_lin(void)
{
    struct port_config pin_conf;
    port_get_config_defaults(&pin_conf);
    pin_conf.direction = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LIN_EN_PIN, &pin_conf);

    /* Enable LIN module*/
    port_pin_set_output_level(LIN_EN_PIN, 1);

    struct usart_config config_lin;
    usart_get_config_defaults(&config_lin);

    /* LIN frame format*/
    config_lin.lin_node = CONF_LIN_NODE_TYPE;
    config_lin.transfer_mode = USART_TRANSFER_ASYNCHRONOUSLY;
    config_lin.sample_rate = USART_SAMPLE_RATE_16X_FRACTIONAL;

    config_lin.baudrate = 115200;
    config_lin.mux_setting = LIN_USART_SERCOM_MUX_SETTING;
    config_lin.pinmux_pad0 = LIN_USART_SERCOM_PINMUX_PAD0;
    config_lin.pinmux_pad1 = LIN_USART_SERCOM_PINMUX_PAD1;
    config_lin.pinmux_pad2 = LIN_USART_SERCOM_PINMUX_PAD2;
    config_lin.pinmux_pad3 = LIN_USART_SERCOM_PINMUX_PAD3;

    /* Disable receiver and transmitter */
    config_lin.receiver_enable = false;
    config_lin.transmitter_enable = false;
}
```

```

if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
    config_lin.lin_slave_enable = true;
}

while (usart_init(&lin_instance,
    LIN_USART_MODULE, &config_lin) != STATUS_OK) {
}

usart_enable(&lin_instance);

usart_register_callback(&lin_instance,
    lin_read_callback, USART_CALLBACK_BUFFER_RECEIVED);
usart_enable_callback(&lin_instance,
    USART_CALLBACK_BUFFER_RECEIVED);
usart_register_callback(&lin_instance,
    lin_read_error_callback, USART_CALLBACK_ERROR);
usart_enable_callback(&lin_instance, USART_CALLBACK_ERROR);
system_interrupt_enable_global();
}

```

Note: The LIN frame format can be configured as master or slave, refer to `CONF_LIN_NODE_TYPE`.

16.8.4.2. Use Case

Code

Copy-paste the following code to your user application:

```

configure_usart_lin();

if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
    printf("LIN Works in Master Mode\r\n");
    if (lin_master_transmission_status(&lin_instance)) {
        usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_TX);
        lin_master_send_cmd(&lin_instance, LIN_MASTER_AUTO_TRANSMIT_CMD);
        usart_write_wait(&lin_instance, LIN_ID_FIELD_VALUE);
        usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_RX);
        while(1) {
            usart_read_buffer_job(&lin_instance,
                (uint8_t *)rx_buffer, 5);
        }
    }
} else {
    printf("LIN Works in Slave Mode\r\n");
    usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_RX);
    while(1) {
        usart_read_buffer_job(&lin_instance,
            (uint8_t *)rx_buffer, 1);
    }
}
}

```

Workflow

1. Set up USART LIN module.

```
configure_usart_lin();
```

2. For LIN master, sending LIN command. For LIN slaver, start reading data .

```

if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
    printf("LIN Works in Master Mode\r\n");
    if (lin_master_transmission_status(&lin_instance)) {

```

```

        usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_TX);

lin_master_send_cmd(&lin_instance, LIN_MASTER_AUTO_TRANSMIT_CMD);
    usart_write_wait(&lin_instance, LIN_ID_FIELD_VALUE);
    usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_RX);
    while(1) {
        usart_read_buffer_job(&lin_instance,
            (uint8_t *)rx_buffer, 5);
    }
}
} else {
    printf("LIN Works in Slave Mode\r\n");
    usart_enable_transceiver(&lin_instance, USART_TRANSCEIVER_RX);
    while(1) {
        usart_read_buffer_job(&lin_instance,
            (uint8_t *)rx_buffer, 1);
    }
}
}

```

16.9. SERCOM USART MUX Settings

The following lists the possible internal SERCOM module pad function assignments, for the four SERCOM pads when in USART mode. Note that this is in addition to the physical GPIO pin MUX of the device, and can be used in conjunction to optimize the serial data pin-out.

When TX and RX are connected to the same pin, the USART will operate in half-duplex mode if both one transmitter and several receivers are enabled.

Note: When RX and XCK are connected to the same pin, the receiver must not be enabled if the USART is configured to use an external clock.

MUX/Pad	PAD 0	PAD 1	PAD 2	PAD 3
RX_0_TX_0_XCK_1	TX / RX	XCK	-	-
RX_0_TX_2_XCK_3	RX	-	TX	XCK
RX_1_TX_0_XCK_1	TX	RX / XCK	-	-
RX_1_TX_2_XCK_3	-	RX	TX	XCK
RX_2_TX_0_XCK_1	TX	XCK	RX	-
RX_2_TX_2_XCK_3	-	-	TX / RX	XCK
RX_3_TX_0_XCK_1	TX	XCK	-	RX
RX_3_TX_2_XCK_3	-	-	TX	RX / XCK

17. SAM I²C Master Mode (SERCOM I²C) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's SERCOM I²C module, for the transfer of data via an I²C bus. The following driver API modes are covered by this manual:

- Master Mode Polled APIs
- Master Mode Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

17.1. Prerequisites

There are no prerequisites.

17.2. Module Overview

The outline of this section is as follows:

- [Driver Feature Macro Definition](#)
- [Functional Description](#)
- [Bus Topology](#)
- [Transactions](#)
- [Multi Master](#)
- [Bus States](#)
- [Bus Timing](#)
- [Operation in Sleep Modes](#)

17.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_I2C_10_BIT_ADDRESS	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_I2C_SCL_STRETCH_MODE	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_I2C_SCL_EXTEND_TIMEOUT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

Note: When using the I2C high-speed mode for off-board communication, there are various high frequency interference, which can lead to distortion of the signals and communication failure. When using Xplained Pro boards in order to test I2C high-speed communication, the following recommendation should be followed:

- Use the SDA-line on PA08 and SCL-line on PA09 for both boards. This will provide stronger pull-ups on both SDA and SCL.
- The SCL should not be higher than 1.5MHz.

17.2.2. Functional Description

The I²C provides a simple two-wire bidirectional bus consisting of a wired-AND type serial clock line (SCL) and a wired-AND type serial data line (SDA).

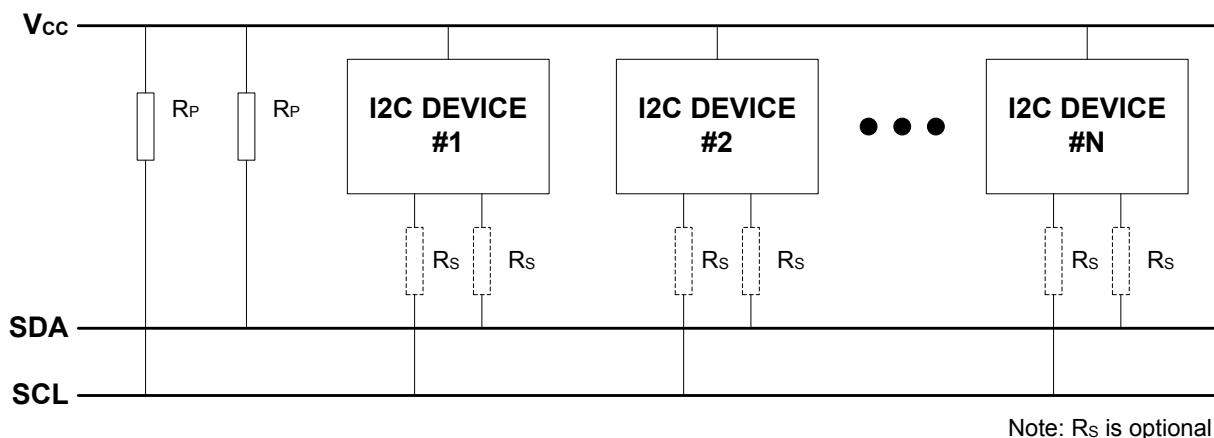
The I²C bus provides a simple, but efficient method of interconnecting multiple master and slave devices. An arbitration mechanism is provided for resolving bus ownership between masters, as only one master device may own the bus at any given time. The arbitration mechanism relies on the wired-AND connections to avoid bus drivers short-circuiting.

A unique address is assigned to all slave devices connected to the bus. A device can contain both master and slave logic, and can emulate multiple slave devices by responding to more than one address.

17.2.3. Bus Topology

The I²C bus topology is illustrated in [Figure 17-1](#). The pull-up resistors (Rs) will provide a high level on the bus lines when none of the I²C devices are driving the bus. These are optional, and can be replaced with a constant current source.

Figure 17-1. I²C Bus Topology



17.2.4. Transactions

The I²C standard defines three fundamental transaction formats:

- Master Write
 - The master transmits data packets to the slave after addressing it
- Master Read
 - The slave transmits data packets to the master after being addressed
- Combined Read/Write
 - A combined transaction consists of several write and read transactions

A data transfer starts with the master issuing a **Start** condition on the bus, followed by the address of the slave together with a bit to indicate whether the master wants to read from or write to the slave. The addressed slave must respond to this by sending an **ACK** back to the master.

After this, data packets are sent from the master or slave, according to the read/write bit. Each packet must be acknowledged (ACK) or not acknowledged (NACK) by the receiver.

If a slave responds with a NACK, the master must assume that the slave cannot receive any more data and cancel the write operation.

The master completes a transaction by issuing a **Stop** condition.

A master can issue multiple **Start** conditions during a transaction; this is then called a **Repeated Start** condition.

17.2.4.1. Address Packets

The slave address consists of seven bits. The 8th bit in the transfer determines the data direction (read or write). An address packet always succeeds a **Start** or **Repeated Start** condition. The 8th bit is handled in the driver, and the user will only have to provide the 7-bit address.

17.2.4.2. Data Packets

Data packets are nine bits long, consisting of one 8-bit data byte, and an acknowledgement bit. Data packets follow either an address packet or another data packet on the bus.

17.2.4.3. Transaction Examples

The gray bits in the following examples are sent from master to slave, and the white bits are sent from slave to master. Example of a read transaction is shown in [Figure 17-2](#). Here, the master first issues a

Start condition and gets ownership of the bus. An address packet with the direction flag set to read is then sent and acknowledged by the slave. Then the slave sends one data packet which is acknowledged by the master. The slave sends another packet, which is not acknowledged by the master and indicates that the master will terminate the transaction. In the end, the transaction is terminated by the master issuing a **Stop** condition.

Figure 17-2. I²C Packet Read

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit 11	Bit 12	Bit 13	Bit 14	Bit 15	Bit 16	Bit 17	Bit 18	Bit 19	Bit 20	Bit 21	Bit 22	Bit 23	Bit 24	Bit 25	Bit 26	Bit 27	Bit 28		
START	ADDRESS							READ	ACK	DATA									ACK	DATA									NACK	STOP

Example of a write transaction is shown in [Figure 17-3](#). Here, the master first issues a **Start** condition and gets ownership of the bus. An address packet with the dir flag set to write is then sent and acknowledged by the slave. Then the master sends two data packets, each acknowledged by the slave. In the end, the transaction is terminated by the master issuing a **Stop** condition.

Figure 17-3. I²C Packet Write

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit 11	Bit 12	Bit 13	Bit 14	Bit 15	Bit 16	Bit 17	Bit 18	Bit 19	Bit 20	Bit 21	Bit 22	Bit 23	Bit 24	Bit 25	Bit 26	Bit 27	Bit 28		
START	ADDRESS							WRITE	ACK	DATA									ACK	DATA									ACK	STOP

17.2.4.4. Packet Timeout

When a master sends an I²C packet, there is no way of being sure that a slave will acknowledge the packet. To avoid stalling the device forever while waiting for an acknowledge, a user selectable timeout is provided in the [i2c_master_config](#) struct which lets the driver exit a read or write operation after the specified time. The function will then return the STATUS_ERR_TIMEOUT flag.

This is also the case for the slave when using the functions postfixed `_wait`.

The time before the timeout occurs, will be the same as for [unknown bus state](#) timeout.

17.2.4.5. Repeated Start

To issue a **Repeated Start**, the functions postfixed `_no_stop` must be used. These functions will not send a **Stop** condition when the transfer is done, thus the next transfer will start with a **Repeated Start**. To end the transaction, the functions without the `_no_stop` postfix must be used for the last read/write.

17.2.5. Multi Master

In a multi master environment, arbitration of the bus is important, as only one master can own the bus at any point.

17.2.5.1. Arbitration

Clock stretching

The serial clock line is always driven by a master device. However, all devices connected to the bus are allowed stretch the low period of the clock to slow down the overall clock frequency or to insert wait states while processing data. Both master and slave can randomly stretch the clock, which will force the other device into a wait-state until the clock line goes high again.

Arbitration on the data line

If two masters start transmitting at the same time, they will both transmit until one master detects that the other master is pulling the data line low. When this is detected, the master not pulling the line low, will stop the transmission and wait until the bus is idle. As it is the master trying to contact the slave with the lowest address that will get the bus ownership, this will create an arbitration scheme always prioritizing the slaves with the lowest address in case of a bus collision.

17.2.5.2. Clock Synchronization

In situations where more than one master is trying to control the bus clock line at the same time, a clock synchronization algorithm based on the same principles used for clock stretching is necessary.

17.2.6. Bus States

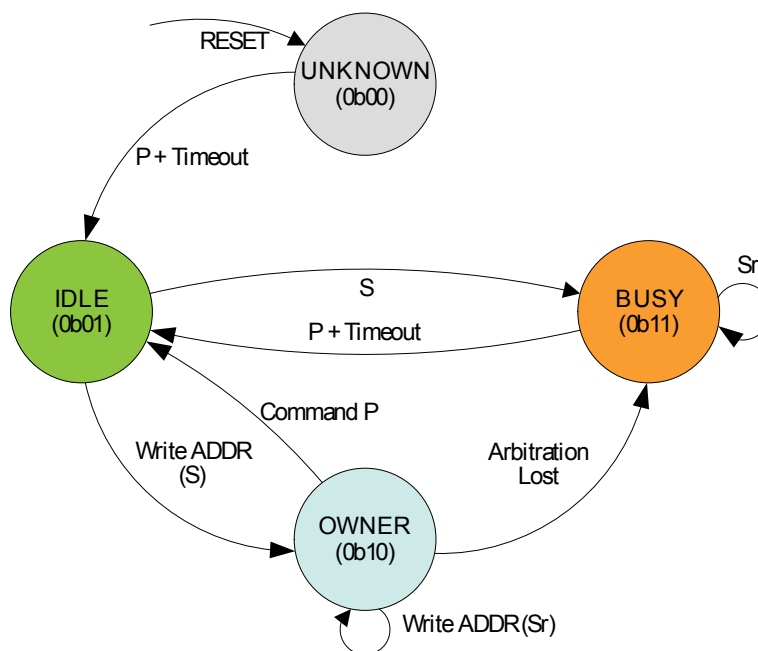
As the I²C bus is limited to one transaction at the time, a master that wants to perform a bus transaction must wait until the bus is free. Because of this, it is necessary for all masters in a multi-master system to know the current status of the bus to be able to avoid conflicts and to ensure data integrity.

- **IDLE** No activity on the bus (between a **Stop** and a new **Start** condition)
- **OWNER** If the master initiates a transaction successfully
- **BUSY** If another master is driving the bus
- **UNKNOWN** If the master has recently been enabled or connected to the bus. Is forced to **IDLE** after given **timeout** when the master module is enabled

The bus state diagram can be seen in [Figure 17-4](#).

- S: Start condition
- P: Stop condition
- Sr: Repeated start condition

Figure 17-4. I²C Bus State Diagram



17.2.7. Bus Timing

Inactive bus timeout for the master and SDA hold time is configurable in the drivers.

17.2.7.1. Unknown Bus State Timeout

When a master is enabled or connected to the bus, the bus state will be unknown until either a given timeout or a stop command has occurred. The timeout is configurable in the `i2c_master_config` struct. The timeout time will depend on toolchain and optimization level used, as the timeout is a loop incrementing a value until it reaches the specified timeout value.

17.2.7.2. SDA Hold Timeout

When using the I²C in slave mode, it will be important to set a SDA hold time which assures that the master will be able to pick up the bit sent from the slave. The SDA hold time makes sure that this is the case by holding the data line low for a given period after the negative edge on the clock.

The SDA hold time is also available for the master driver, but is not a necessity.

17.2.8. Operation in Sleep Modes

The I²C module can operate in all sleep modes by setting the `run_in_standby` Boolean in the `i2c_master_config` or `i2c_slave_config` struct. The operation in slave and master mode is shown in [Table 17-1](#).

Table 17-1. I²C Standby Operations

Run in standby	Slave	Master
false	Disabled, all reception is dropped	Generic Clock (GCLK) disabled when master is idle
true	Wake on address match when enabled	GCLK enabled while in sleep modes

17.3. Special Considerations

17.3.1. Interrupt-driven Operation

While an interrupt-driven operation is in progress, subsequent calls to a write or read operation will return the `STATUS_BUSY` flag, indicating that only one operation is allowed at any given time.

To check if another transmission can be initiated, the user can either call another transfer operation, or use the `i2c_master_get_job_status`/`i2c_slave_get_job_status` functions depending on mode.

If the user would like to get callback from operations while using the interrupt-driven driver, the callback must be registered and then enabled using the "register_callback" and "enable_callback" functions.

17.4. Extra Information

For extra information, see [Extra Information for SERCOM I2C Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

17.5. Examples

For a list of examples related to this driver, see [Examples for SERCOM I2C Driver](#).

17.6. API Overview

17.6.1. Structure Definitions

17.6.1.1. Struct `i2c_master_config`

This is the configuration structure for the I²C Master device. It is used as an argument for [i2c_master_init](#) to provide the desired configurations for the module. The structure should be initialized using the [i2c_master_get_config_defaults](#).

Table 17-2. Members

Type	Name	Description
uint32_t	baud_rate	Baud rate (in KHz) for I ² C operations in standard-mode, Fast-mode, and Fast-mode Plus Transfers, i2c_master_baud_rate
uint32_t	baud_rate_high_speed	Baud rate (in KHz) for I ² C operations in High-speed mode, i2c_master_baud_rate
uint16_t	buffer_timeout	Timeout for packet write to wait for slave
enum gclk_generator	generator_source	GCLK generator to use as clock source
enum i2c_master_inactive_timeout	inactive_timeout	Inactive bus time out
bool	master_scl_low_extend_timeout	Set to enable maser SCL low extend time-out
uint32_t	pinmux_pad0	PAD0 (SDA) pinmux
uint32_t	pinmux_pad1	PAD1 (SCL) pinmux
bool	run_in_standby	Set to keep module active in sleep modes
bool	scl_low_timeout	Set to enable SCL low time-out
bool	scl_stretch_only_after_ack_bit	Set to enable SCL stretch only after ACK bit (required for high speed)
uint16_t	sda_scl_rise_time_ns	Get more accurate BAUD, considering rise time(required for standard-mode and Fast-mode)
bool	slave_scl_low_extend_timeout	Set to enable slave SCL low extend time-out

Type	Name	Description
enum i2c_master_start_hold_time	start_hold_time	Bus hold time after start signal on data line
enum i2c_master_transfer_speed	transfer_speed	Transfer speed mode
uint16_t	unknown_bus_state_timeout	Unknown bus state timeout

17.6.1.2. Struct i2c_master_module

SERCOM I²C Master driver software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

17.6.1.3. Struct i2c_master_packet

Structure to be used when transferring I²C master packets.

Table 17-3. Members

Type	Name	Description
uint16_t	address	Address to slave device
uint8_t *	data	Data array containing all data to be transferred
uint16_t	data_length	Length of data array
bool	high_speed	Use high speed transfer. Set to false if the feature is not supported by the device
uint8_t	hs_master_code	High speed mode master code (0000 1XXX), valid when high_speed is true
bool	ten_bit_address	Use 10-bit addressing. Set to false if the feature is not supported by the device

17.6.2. Macro Definitions

17.6.2.1. Driver Feature Definition

Define SERCOM I²C driver features set according to different device family.

Macro FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED

```
#define FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED
```

Fast mode plus and high speed support.

Macro FEATURE_I2C_10_BIT_ADDRESS

```
#define FEATURE_I2C_10_BIT_ADDRESS
```

10-bit address support

Macro FEATURE_I2C_SCL_STRETCH_MODE

```
#define FEATURE_I2C_SCL_STRETCH_MODE
```

SCL stretch mode support

Macro FEATURE_I2C_SCL_EXTEND_TIMEOUT

```
#define FEATURE_I2C_SCL_EXTEND_TIMEOUT
```

SCL extend timeout support

Macro FEATURE_I2C_DMA_SUPPORT

```
#define FEATURE_I2C_DMA_SUPPORT
```

17.6.3. Function Definitions

17.6.3.1. Lock/Unlock

Function i2c_master_lock()

Attempt to get lock on driver instance.

```
enum status_code i2c_master_lock(  
    struct i2c_master_module *const module)
```

This function checks the instance's lock, which indicates whether or not it is currently in use, and sets the lock if it was not already set.

The purpose of this is to enable exclusive access to driver instances, so that, e.g., transactions by different services will not interfere with each other.

Table 17-4. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 17-5. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

Function i2c_master_unlock()

Unlock driver instance.

```
void i2c_master_unlock(  
    struct i2c_master_module *const module)
```

This function clears the instance lock, indicating that it is available for use.

Table 17-6. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 17-7. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

17.6.3.2. Configuration and Initialization

Function i2c_master_is_syncing()

Returns the synchronization status of the module.

```
bool i2c_master_is_syncing(
    const struct i2c_master_module *const module)
```

Returns the synchronization status of the module.

Table 17-8. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure

Returns

Status of the synchronization.

Table 17-9. Return Values

Return value	Description
true	Module is busy synchronizing
false	Module is not synchronizing

Function i2c_master_get_config_defaults()

Gets the I²C master default configurations.

```
void i2c_master_get_config_defaults(
    struct i2c_master_config *const config)
```

Use to initialize the configuration structure to known default values.

The default configuration is as follows:

- Baudrate 100KHz
- GCLK generator 0
- Do not run in standby
- Start bit hold time 300ns - 600ns
- Buffer timeout = 65535
- Unknown bus status timeout = 65535
- Do not run in standby
- PINMUX_DEFAULT for SERCOM pads

Those default configuration only available if the device supports it:

- High speed baudrate 3.4MHz

- Standard-mode and Fast-mode transfer speed
- SCL stretch disabled
- Slave SCL low extend time-out disabled
- Master SCL low extend time-out disabled

Table 17-10. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to configuration structure to be initiated

Function `i2c_master_init()`

Initializes the requested I²C hardware module.

```
enum status_code i2c_master_init(
    struct i2c_master_module *const module,
    Sercom *const hw,
    const struct i2c_master_config *const config)
```

Initializes the SERCOM I²C master device requested and sets the provided software module struct. Run this function before any further use of the driver.

Table 17-11. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to software module struct
[in]	hw	Pointer to the hardware instance
[in]	config	Pointer to the configuration struct

Returns

Status of initialization.

Table 17-12. Return Values

Return value	Description
STATUS_OK	Module initiated correctly
STATUS_ERR_DENIED	If module is enabled
STATUS_BUSY	If module is busy resetting
STATUS_ERR_ALREADY_INITIALIZED	If setting other GCLK generator than previously set
STATUS_ERR_BAUDRATE_UNAVAILABLE	If given baudrate is not compatible with set GCLK frequency

Function `i2c_master_enable()`

Enables the I²C module.

```
void i2c_master_enable(
    const struct i2c_master_module *const module)
```

Enables the requested I²C module and set the bus state to IDLE after the specified [timeout](#) period if no stop bit is detected.

Table 17-13. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software module struct

Function i2c_master_disable()

Disable the I²C module.

```
void i2c_master_disable(  
    const struct i2c_master_module *const module)
```

Disables the requested I²C module.

Table 17-14. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software module struct

Function i2c_master_reset()

Resets the hardware module.

```
void i2c_master_reset(  
    struct i2c_master_module *const module)
```

Reset the module to hardware defaults.

Table 17-15. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

17.6.3.3. Read and Write

Function i2c_master_read_packet_wait()

Reads data packet from slave.

```
enum status_code i2c_master_read_packet_wait(  
    struct i2c_master_module *const module,  
    struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I²C bus and sends a stop condition when finished.

Note: This will stall the device from any other operation. For interrupt-driven operation, see [i2c_master_read_packet_job](#).

Table 17-16. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of reading packet.

Table 17-17. Return Values

Return value	Description
STATUS_OK	The packet was read successfully
STATUS_ERR_TIMEOUT	If no response was given within specified timeout period
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address

Function i2c_master_read_packet_wait_no_stop()

Reads data packet from slave without sending a stop condition when done.

```
enum status_code i2c_master_read_packet_wait_no_stop(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I²C bus without sending a stop condition when done, thus retaining ownership of the bus when done. To end the transaction, a [read](#) or [write](#) with stop condition must be performed.

Note: This will stall the device from any other operation. For interrupt-driven operation, see [i2c_master_read_packet_job](#).

Table 17-18. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of reading packet.

Table 17-19. Return Values

Return value	Description
STATUS_OK	The packet was read successfully
STATUS_ERR_TIMEOUT	If no response was given within specified timeout period

Return value	Description
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address

Function `i2c_master_write_packet_wait()`

Writes data packet to slave.

```
enum status_code i2c_master_write_packet_wait(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I²C bus and sends a stop condition when finished.

Note: This will stall the device from any other operation. For interrupt-driven operation, see [i2c_master_read_packet_job](#).

Table 17-20. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of write packet.

Table 17-21. Return Values

Return value	Description
STATUS_OK	If packet was write successfully
STATUS_BUSY	If master module is busy with a job
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address
STATUS_ERR_TIMEOUT	If timeout occurred
STATUS_ERR_OVERFLOW	If slave did not acknowledge last sent data, indicating that slave does not want more data and was not able to read last data sent

Function `i2c_master_write_packet_wait_no_stop()`

Writes data packet to slave without sending a stop condition when done.

```
enum status_code i2c_master_write_packet_wait_no_stop(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I²C bus without sending a stop condition, thus retaining ownership of the bus when done. To end the transaction, a [read](#) or [write](#) with stop condition or sending a stop with the [i2c_master_send_stop](#) function must be performed.

Note: This will stall the device from any other operation. For interrupt-driven operation, see [i2c_master_read_packet_job](#).

Table 17-22. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of write packet.

Table 17-23. Return Values

Return value	Description
STATUS_OK	If packet was write successfully
STATUS_BUSY	If master module is busy
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address
STATUS_ERR_TIMEOUT	If timeout occurred
STATUS_ERR_OVERFLOW	If slave did not acknowledge last sent data, indicating that slave do not want more data

Function i2c_master_send_stop()

Sends stop condition on bus.

```
void i2c_master_send_stop(
    struct i2c_master_module *const module)
```

Sends a stop condition on bus.

Note: This function can only be used after the [i2c_master_write_packet_wait_no_stop](#) function. If a stop condition is to be sent after a read, the [i2c_master_read_packet_wait](#) function must be used.

Table 17-24. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `i2c_master_send_nack()`

Sends nack signal on bus.

```
void i2c_master_send_nack(  
    struct i2c_master_module *const module)
```

Sends a nack signal on bus.

Note: This function can only be used after the `i2c_master_write_packet_wait_no_nack` function, or `i2c_master_read_byte` function.

Table 17-25. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `i2c_master_read_byte()`

Reads one byte data from slave.

```
enum status_code i2c_master_read_byte(  
    struct i2c_master_module *const module,  
    uint8_t * byte)
```

Table 17-26. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[out]	byte	Read one byte data to slave

Returns

Status of reading byte.

Table 17-27. Return Values

Return value	Description
STATUS_OK	One byte was read successfully
STATUS_ERR_TIMEOUT	If no response was given within specified timeout period
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address

Function `i2c_master_write_byte()`

Write one byte data to slave.

```
enum status_code i2c_master_write_byte(  
    struct i2c_master_module *const module,  
    uint8_t byte)
```

Table 17-28. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in]	byte	Send one byte data to slave

Returns

Status of writing byte.

Table 17-29. Return Values

Return value	Description
STATUS_OK	One byte was write successfully
STATUS_ERR_TIMEOUT	If no response was given within specified timeout period
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address

Function i2c_master_read_packet_wait_no_nack()

```
enum status_code i2c_master_read_packet_wait_no_nack(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

17.6.3.4. SERCOM I²C Master with DMA Interfaces**Function i2c_master_dma_set_transfer()**

Set I²C for DMA transfer with slave address and transfer size.

```
void i2c_master_dma_set_transfer(
    struct i2c_master_module *const module,
    uint16_t addr,
    uint8_t length,
    enum i2c_transfer_direction direction)
```

This function will set the slave address, transfer size and enable the auto transfer mode for DMA.

Table 17-30. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock
[in]	addr	I ² C slave address
[in]	length	I ² C transfer length with DMA
[in]	direction	I ² C transfer direction

17.6.3.5. Callbacks

Function `i2c_master_register_callback()`

Registers callback for the specified callback type.

```
void i2c_master_register_callback(  
    struct i2c_master_module *const module,  
    i2c_master_callback_t callback,  
    enum i2c_master_callback callback_type)
```

Associates the given callback function with the specified callback type.

To enable the callback, the `i2c_master_enable_callback` function must be used.

Table 17-31. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback	Pointer to the function desired for the specified callback
[in]	callback_type	Callback type to register

Function `i2c_master_unregister_callback()`

Unregisters callback for the specified callback type.

```
void i2c_master_unregister_callback(  
    struct i2c_master_module *const module,  
    enum i2c_master_callback callback_type)
```

When called, the currently registered callback for the given callback type will be removed.

Table 17-32. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Specifies the callback type to unregister

Function `i2c_master_enable_callback()`

Enables callback.

```
void i2c_master_enable_callback(  
    struct i2c_master_module *const module,  
    enum i2c_master_callback callback_type)
```

Enables the callback specified by the `callback_type`.

Table 17-33. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Callback type to enable

Function `i2c_master_disable_callback()`

Disables callback.

```
void i2c_master_disable_callback(  
    struct i2c_master_module *const module,  
    enum i2c_master_callback callback_type)
```

Disables the callback specified by the `callback_type`.

Table 17-34. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Callback type to disable

17.6.3.6. Read and Write, Interrupt-driven

Function `i2c_master_read_bytes()`

```
enum status_code i2c_master_read_bytes(  
    struct i2c_master_module *const module,  
    struct i2c_master_packet *const packet)
```

Function `i2c_master_read_packet_job()`

Initiates a read packet operation.

```
enum status_code i2c_master_read_packet_job(  
    struct i2c_master_module *const module,  
    struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I²C bus. This is the non-blocking equivalent of [i2c_master_read_packet_wait](#).

Table 17-35. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting reading I²C packet.

Table 17-36. Return Values

Return value	Description
STATUS_OK	If reading was started successfully
STATUS_BUSY	If module is currently busy with another transfer

Function `i2c_master_read_packet_job_no_stop()`

Initiates a read packet operation without sending a STOP condition when done.

```
enum status_code i2c_master_read_packet_job_no_stop(  
    struct i2c_master_module *const module,  
    struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I²C bus without sending a stop condition, thus retaining ownership of the bus when done. To end the transaction, a [read](#) or [write](#) with stop condition must be performed.

This is the non-blocking equivalent of [i2c_master_read_packet_wait_no_stop](#).

Table 17-37. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting reading I²C packet.

Table 17-38. Return Values

Return value	Description
STATUS_OK	If reading was started successfully
STATUS_BUSY	If module is currently busy with another operation

Function `i2c_master_read_packet_job_no_nack()`

Initiates a read packet operation without sending a NACK signal and a STOP condition when done.

```
enum status_code i2c_master_read_packet_job_no_nack(  
    struct i2c_master_module *const module,  
    struct i2c_master_packet *const packet)
```

Reads a data packet from the specified slave address on the I²C bus without sending a nack and a stop condition, thus retaining ownership of the bus when done. To end the transaction, a [read](#) or [write](#) with stop condition must be performed.

This is the non-blocking equivalent of [i2c_master_read_packet_wait_no_stop](#).

Table 17-39. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting reading I²C packet.

Table 17-40. Return Values

Return value	Description
STATUS_OK	If reading was started successfully
STATUS_BUSY	If module is currently busy with another operation

Function i2c_master_write_bytes()

```
enum status_code i2c_master_write_bytes(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

Function i2c_master_write_packet_job()

Initiates a write packet operation.

```
enum status_code i2c_master_write_packet_job(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I²C bus. This is the non-blocking equivalent of [i2c_master_write_packet_wait](#).

Table 17-41. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting writing I²C packet job.

Table 17-42. Return Values

Return value	Description
STATUS_OK	If writing was started successfully
STATUS_BUSY	If module is currently busy with another transfer

Function i2c_master_write_packet_job_no_stop()

Initiates a write packet operation without sending a STOP condition when done.

```
enum status_code i2c_master_write_packet_job_no_stop(
    struct i2c_master_module *const module,
    struct i2c_master_packet *const packet)
```

Writes a data packet to the specified slave address on the I²C bus without sending a stop condition, thus retaining ownership of the bus when done. To end the transaction, a [read](#) or [write](#) with stop condition or sending a stop with the [i2c_master_send_stop](#) function must be performed.

This is the non-blocking equivalent of [i2c_master_write_packet_wait_no_stop](#).

Table 17-43. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting writing I²C packet job.

Table 17-44. Return Values

Return value	Description
STATUS_OK	If writing was started successfully
STATUS_BUSY	If module is currently busy with another

Function i2c_master_cancel_job()

Cancel any currently ongoing operation.

```
void i2c_master_cancel_job(
    struct i2c_master_module *const module)
```

Terminates the running transfer operation.

Table 17-45. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

Function i2c_master_get_job_status()

Get status from ongoing job.

```
enum status_code i2c_master_get_job_status(
    struct i2c_master_module *const module)
```

Will return the status of a transfer operation.

Table 17-46. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure

Returns

Last status code from transfer operation.

Table 17-47. Return Values

Return value	Description
STATUS_OK	No error has occurred
STATUS_BUSY	If transfer is in progress

Return value	Description
STATUS_BUSY	If master module is busy
STATUS_ERR_DENIED	If error on bus
STATUS_ERR_PACKET_COLLISION	If arbitration is lost
STATUS_ERR_BAD_ADDRESS	If slave is busy, or no slave acknowledged the address
STATUS_ERR_TIMEOUT	If timeout occurred
STATUS_ERR_OVERFLOW	If slave did not acknowledge last sent data, indicating that slave does not want more data and was not able to read

17.6.4. Enumeration Definitions

17.6.4.1. Enum i2c_master_baud_rate

Values for I²C speeds supported by the module. The driver will also support setting any other value, in which case set the value in the [i2c_master_config](#) at desired value divided by 1000.

Example: If 10KHz operation is required, give baud_rate in the configuration structure the value 10.

Table 17-48. Members

Enum value	Description
I2C_MASTER_BAUD_RATE_100KHZ	Baud rate at 100KHz (Standard-mode)
I2C_MASTER_BAUD_RATE_400KHZ	Baud rate at 400KHz (Fast-mode)
I2C_MASTER_BAUD_RATE_1000KHZ	Baud rate at 1MHz (Fast-mode Plus)
I2C_MASTER_BAUD_RATE_3400KHZ	Baud rate at 3.4MHz (High-speed mode)

17.6.4.2. Enum i2c_master_callback

The available callback types for the I²C master module.

Table 17-49. Members

Enum value	Description
I2C_MASTER_CALLBACK_WRITE_COMPLETE	Callback for packet write complete
I2C_MASTER_CALLBACK_READ_COMPLETE	Callback for packet read complete
I2C_MASTER_CALLBACK_ERROR	Callback for error

17.6.4.3. Enum i2c_master_inactive_timeout

If the inactive bus time-out is enabled and the bus is inactive for longer than the time-out setting, the bus state logic will be set to idle.

Table 17-50. Members

Enum value	Description
I2C_MASTER_INACTIVE_TIMEOUT_DISABLED	Inactive bus time-out disabled
I2C_MASTER_INACTIVE_TIMEOUT_55US	Inactive bus time-out 5-6 SCL cycle time-out
I2C_MASTER_INACTIVE_TIMEOUT_105US	Inactive bus time-out 10-11 SCL cycle time-out
I2C_MASTER_INACTIVE_TIMEOUT_205US	Inactive bus time-out 20-21 SCL cycle time-out

17.6.4.4. Enum i2c_master_interrupt_flag

Flags used when reading or setting interrupt flags.

Table 17-51. Members

Enum value	Description
I2C_MASTER_INTERRUPT_WRITE	Interrupt flag used for write
I2C_MASTER_INTERRUPT_READ	Interrupt flag used for read

17.6.4.5. Enum i2c_master_start_hold_time

Values for the possible I²C master mode SDA internal hold times after start bit has been sent.

Table 17-52. Members

Enum value	Description
I2C_MASTER_START_HOLD_TIME_DISABLED	Internal SDA hold time disabled
I2C_MASTER_START_HOLD_TIME_50NS_100NS	Internal SDA hold time 50ns - 100ns
I2C_MASTER_START_HOLD_TIME_300NS_600NS	Internal SDA hold time 300ns - 600ns
I2C_MASTER_START_HOLD_TIME_400NS_800NS	Internal SDA hold time 400ns - 800ns

17.6.4.6. Enum i2c_master_transfer_speed

Enum for the transfer speed.

Table 17-53. Members

Enum value	Description
I2C_MASTER_SPEED_STANDARD_AND_FAST	Standard-mode (Sm) up to 100KHz and Fast-mode (Fm) up to 400KHz
I2C_MASTER_SPEED_FAST_MODE_PLUS	Fast-mode Plus (Fm+) up to 1MHz
I2C_MASTER_SPEED_HIGH_SPEED	High-speed mode (Hs-mode) up to 3.4MHz

17.6.4.7. Enum i2c_transfer_direction

For master: transfer direction or setting direction bit in address. For slave: direction of request from master.

Table 17-54. Members

Enum value	Description
I2C_TRANSFER_WRITE	Master write operation is in progress
I2C_TRANSFER_READ	Master read operation is in progress

17.7. Extra Information for SERCOM I²C Driver

17.7.1. Acronyms

Table 17-55 is a table listing the acronyms used in this module, along with their intended meanings.

Table 17-55. Acronyms

Acronym	Description
SDA	Serial Data Line
SCL	Serial Clock Line
SERCOM	Serial Communication Interface
DMA	Direct Memory Access

17.7.2. Dependencies

The I²C driver has the following dependencies:

- System Pin Multiplexer Driver

17.7.3. Errata

There are no errata related to this driver.

17.7.4. Module History

Table 17-56 is an overview of the module history, detailing enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version listed in Table 17-56.

Table 17-56. Module History

Changelog
• Added 10-bit addressing and high speed support in SAM D21 • Separate structure i2c_packet into i2c_master_packet and i2c_slave packet
• Added support for SCL stretch and extended timeout hardware features in SAM D21 • Added fast mode plus support in SAM D21
Fixed incorrect logical mask for determining if a bus error has occurred in I ² C Slave mode
Initial Release

17.8. Examples for SERCOM I²C Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM I2C Master Mode \(SERCOM I2C\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for the I2C Master module - Basic Use Case](#)
 - [Quick Start Guide for the I2C Master module - Callback Use Case](#)
- [Quick Start Guide for the I2C Master module - DMA Use Case](#)

17.8.1. Quick Start Guide for SERCOM I²C Master - Basic

In this use case, the I²C will be used and set up as follows:

- Master mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value
- 65535 unknown bus state timeout value

17.8.1.1. Prerequisites

The device must be connected to an I²C slave.

17.8.1.2. Setup

Code

The following must be added to the user application:

- A sample buffer to send, a sample buffer to read:

```
#define DATA_LENGTH 10
static uint8_t write_buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
};

static uint8_t read_buffer[DATA_LENGTH];
```

- Slave address to access:

```
#define SLAVE_ADDRESS 0x12
```

- Number of times to try to send packet if it fails:

```
#define TIMEOUT 1000
```

- Globally accessible module structure:

```
struct i2c_master_module i2c_master_instance;
```

- Function for setting up the module:

```
void configure_i2c_master(void)
{
    /* Initialize config structure and software module. */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer. */
    config_i2c_master.buffer_timeout = 10000;
```

```

#if SAMR30
    config_i2c_master.pinmux_pad0    = CONF_MASTER_SDA_PINMUX;
    config_i2c_master.pinmux_pad1    = CONF_MASTER_SCK_PINMUX;
#endif
    /* Initialize and enable device with config. */
    i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}

```

- Add to user application main():

```

/* Configure device and enable. */
configure_i2c_master();

/* Timeout counter. */
uint16_t timeout = 0;

/* Init i2c packet. */
struct i2c_master_packet packet = {
    .address      = SLAVE_ADDRESS,
    .data_length  = DATA_LENGTH,
    .data         = write_buffer,
    .ten_bit_address = false,
    .high_speed   = false,
    .hs_master_code = 0x0,
};

```

Workflow

1. Configure and enable module.

```

void configure_i2c_master(void)
{
    /* Initialize config structure and software module. */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer. */
    config_i2c_master.buffer_timeout = 10000;

    /* Initialize and enable device with config. */
    i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}

```

1. Create and initialize configuration structure.

```

struct i2c_master_config config_i2c_master;
i2c_master_get_config_defaults(&config_i2c_master);

```

2. Change settings in the configuration.

```

config_i2c_master.buffer_timeout = 10000;

```

3. Initialize the module with the set configurations.

```

i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
&config_i2c_master);

```

4. Enable the module.

```
i2c_master_enable(&i2c_master_instance);
```

2. Create a variable to see when we should stop trying to send packet.

```
uint16_t timeout = 0;
```

3. Create a packet to send.

```
struct i2c_master_packet packet = {  
    .address      = SLAVE_ADDRESS,  
    .data_length  = DATA_LENGTH,  
    .data         = write_buffer,  
    .ten_bit_address = false,  
    .high_speed   = false,  
    .hs_master_code = 0x0,  
};
```

17.8.1.3. Implementation

Code

Add to user application main():

```
/* Write buffer to slave until success. */  
while (i2c_master_write_packet_wait(&i2c_master_instance, &packet) !=  
    STATUS_OK) {  
    /* Increment timeout counter and check if timed out. */  
    if (timeout++ == TIMEOUT) {  
        break;  
    }  
}  
  
/* Read from slave until success. */  
packet.data = read_buffer;  
while (i2c_master_read_packet_wait(&i2c_master_instance, &packet) !=  
    STATUS_OK) {  
    /* Increment timeout counter and check if timed out. */  
    if (timeout++ == TIMEOUT) {  
        break;  
    }  
}
```

Workflow

1. Write packet to slave.

```
while (i2c_master_write_packet_wait(&i2c_master_instance, &packet) !=  
    STATUS_OK) {  
    /* Increment timeout counter and check if timed out. */  
    if (timeout++ == TIMEOUT) {  
        break;  
    }  
}
```

The module will try to send the packet TIMEOUT number of times or until it is successfully sent.

2. Read packet from slave.

```
packet.data = read_buffer;  
while (i2c_master_read_packet_wait(&i2c_master_instance, &packet) !=  
    STATUS_OK) {  
    /* Increment timeout counter and check if timed out. */
```

```

        if (timeout++ == TIMEOUT) {
            break;
        }
    }
}

```

The module will try to read the packet TIMEOUT number of times or until it is successfully read.

17.8.2. Quick Start Guide for SERCOM I²C Master - Callback

In this use case, the I²C will be used and set up as follows:

- Master mode
- 100KHz operation speed
- Not operational in standby
- 65535 unknown bus state timeout value

17.8.2.1. Prerequisites

The device must be connected to an I²C slave.

17.8.2.2. Setup

Code

The following must be added to the user application:

A sample buffer to write from, a reversed buffer to write from and length of buffers.

```

#define DATA_LENGTH 8

static uint8_t wr_buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07
};

static uint8_t wr_buffer_reversed[DATA_LENGTH] = {
    0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00
};

static uint8_t rd_buffer[DATA_LENGTH];

```

Address of slave:

```

#define SLAVE_ADDRESS 0x12

```

Globally accessible module structure:

```

struct i2c_master_module i2c_master_instance;

```

Globally accessible packet:

```

struct i2c_master_packet wr_packet;
struct i2c_master_packet rd_packet;

```

Function for setting up module:

```

void configure_i2c(void)
{
    /* Initialize config structure and software module */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer */
    config_i2c_master.buffer_timeout = 65535;
#ifdef SAMR30

```

```

    config_i2c_master.pinmux_pad0    = CONF_MASTER_SDA_PINMUX;
    config_i2c_master.pinmux_pad1    = CONF_MASTER_SCK_PINMUX;
#endif

    /* Initialize and enable device with config */
    while(i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
    &config_i2c_master) \
        != STATUS_OK);

    i2c_master_enable(&i2c_master_instance);
}

```

Callback function for write complete:

```

void i2c_write_complete_callback(
    struct i2c_master_module *const module)
{
    /* Initiate new packet read */
    i2c_master_read_packet_job(&i2c_master_instance, &rd_packet);
}

```

Function for setting up the callback functionality of the driver:

```

void configure_i2c_callbacks(void)
{
    /* Register callback function. */
    i2c_master_register_callback(&i2c_master_instance,
    i2c_write_complete_callback,
        I2C_MASTER_CALLBACK_WRITE_COMPLETE);
    i2c_master_enable_callback(&i2c_master_instance,
        I2C_MASTER_CALLBACK_WRITE_COMPLETE);
}

```

Add to user application main():

```

/* Configure device and enable. */
configure_i2c();
/* Configure callbacks and enable. */
configure_i2c_callbacks();

```

Workflow

1. Configure and enable module.

```
configure_i2c();
```

1. Create and initialize configuration structure.

```

struct i2c_master_config config_i2c_master;
i2c_master_get_config_defaults(&config_i2c_master);

```

2. Change settings in the configuration.

```
config_i2c_master.buffer_timeout = 65535;
```

3. Initialize the module with the set configurations.

```

while(i2c_master_init(&i2c_master_instance,
    CONF_I2C_MASTER_MODULE, &config_i2c_master) \
    != STATUS_OK);

```

4. Enable the module.

```
i2c_master_enable(&i2c_master_instance);
```

2. Configure callback functionality.

```
configure_i2c_callbacks();
```

1. Register write complete callback.

```
i2c_master_register_callback(&i2c_master_instance,  
i2c_write_complete_callback,  
I2C_MASTER_CALLBACK_WRITE_COMPLETE);
```

2. Enable write complete callback.

```
i2c_master_enable_callback(&i2c_master_instance,  
I2C_MASTER_CALLBACK_WRITE_COMPLETE);
```

3. Create a packet to send to slave.

```
wr_packet.address      = SLAVE_ADDRESS;  
wr_packet.data_length  = DATA_LENGTH;  
wr_packet.data         = wr_buffer;
```

17.8.2.3. Implementation

Code

Add to user application main():

```
while (true) {  
    /* Infinite loop */  
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {  
        while (!port_pin_get_input_level(BUTTON_0_PIN)) {  
            /* Waiting for button steady */  
        }  
        /* Send every other packet with reversed data */  
        if (wr_packet.data[0] == 0x00) {  
            wr_packet.data = &wr_buffer_reversed[0];  
        } else {  
            wr_packet.data = &wr_buffer[0];  
        }  
        i2c_master_write_packet_job(&i2c_master_instance, &wr_packet);  
    }  
}
```

Workflow

1. Write packet to slave.

```
wr_packet.address      = SLAVE_ADDRESS;  
wr_packet.data_length  = DATA_LENGTH;  
wr_packet.data         = wr_buffer;
```

2. Infinite while loop, while waiting for interaction with slave.

```
while (true) {  
    /* Infinite loop */  
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {  
        while (!port_pin_get_input_level(BUTTON_0_PIN)) {  
            /* Waiting for button steady */  
        }  
        /* Send every other packet with reversed data */  
        if (wr_packet.data[0] == 0x00) {  
            wr_packet.data = &wr_buffer_reversed[0];  
        } else {  
            wr_packet.data = &wr_buffer[0];  
        }  
    }  
}
```

```

        i2c_master_write_packet_job(&i2c_master_instance, &wr_packet);
    }
}

```

17.8.2.4. Callback

Each time a packet is sent, the callback function will be called.

Workflow

- Write complete callback:

1. Send every other packet in reversed order.

```

if (wr_packet.data[0] == 0x00) {
    wr_packet.data = &wr_buffer_reversed[0];
} else {
    wr_packet.data = &wr_buffer[0];
}

```

2. Write new packet to slave.

```

wr_packet.address      = SLAVE_ADDRESS;
wr_packet.data_length  = DATA_LENGTH;
wr_packet.data         = wr_buffer;

```

17.8.3. Quick Start Guide for Using DMA with SERCOM I²C Master

The supported board list:

- SAMD21 Xplained Pro
- SAMR21 Xplained Pro
- SAML21 Xplained Pro
- SAML22 Xplained Pro
- SAMDA1 Xplained Pro
- SAMC21 Xplained Pro

In this use case, the I²C will be used and set up as follows:

- Master mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value
- 65535 unknown bus state timeout value

17.8.3.1. Prerequisites

The device must be connected to an I²C slave.

17.8.3.2. Setup

Code

The following must be added to the user application:

- A sample buffer to send, number of entries to send and address of slave:

```

#define DATA_LENGTH 10
static uint8_t buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
};

#define SLAVE_ADDRESS 0x12

```

Number of times to try to send packet if it fails:

```
#define TIMEOUT 1000
```

- Globally accessible module structure:

```
struct i2c_master_module i2c_master_instance;
```

- Function for setting up the module:

```
static void configure_i2c_master(void)
{
    /* Initialize config structure and software module. */
    struct i2c_master_config config_i2c_master;
    i2c_master_get_config_defaults(&config_i2c_master);

    /* Change buffer timeout to something longer. */
    config_i2c_master.buffer_timeout = 10000;
#ifdef SAMR30
    config_i2c_master.pinmux_pad0 = CONF_MASTER_SDA_PINMUX;
    config_i2c_master.pinmux_pad1 = CONF_MASTER_SCK_PINMUX;
#endif

    /* Initialize and enable device with config. */
    i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
    &config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}
```

- Globally accessible DMA module structure:

```
struct dma_resource example_resource;
```

- Globally transfer done flag:

```
static volatile bool transfer_is_done = false;
```

- Globally accessible DMA transfer descriptor:

```
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor SECTION_DMAC_DESCRIPTOR;
```

- Function for transfer done callback:

```
static void transfer_done(struct dma_resource* const resource )
{
    UNUSED(resource);

    transfer_is_done = true;
}
```

- Function for setting up the DMA resource:

```
static void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.peripheral_trigger = CONF_I2C_DMA_TRIGGER;
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(resource, &config);
}
```


- Function for setting up the DMA transfer descriptor:

```
static void setup_dma_descriptor(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.block_transfer_count = DATA_LENGTH;
    descriptor_config.source_address = (uint32_t)buffer + DATA_LENGTH;
    descriptor_config.destination_address =
        (uint32_t)(&i2c_master_instance.hw->I2CM.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}
```

- Add to user application main():

```
configure_i2c_master();

configure_dma_resource(&example_resource);
setup_dma_descriptor(&example_descriptor);
dma_add_descriptor(&example_resource, &example_descriptor);
dma_register_callback(&example_resource, transfer_done,
    DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource, DMA_CALLBACK_TRANSFER_DONE);
```

Workflow

1. Configure and enable module:

```
configure_i2c_master();
```

1. Create and initialize configuration structure.

```
struct i2c_master_config config_i2c_master;
i2c_master_get_config_defaults(&config_i2c_master);
```

2. Change settings in the configuration.

```
config_i2c_master.buffer_timeout = 10000;
#if SAMR30
config_i2c_master.pinmux_pad0 = CONF_MASTER_SDA_PINMUX;
config_i2c_master.pinmux_pad1 = CONF_MASTER_SCK_PINMUX;
#endif
```

3. Initialize the module with the set configurations.

```
i2c_master_init(&i2c_master_instance, CONF_I2C_MASTER_MODULE,
    &config_i2c_master);
```

4. Enable the module.

```
i2c_master_enable(&i2c_master_instance);
```

2. Configure DMA

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

2. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM TX trigger causes a transaction transfer in this example.

```
config.peripheral_trigger = CONF_I2C_DMA_TRIGGER;  
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

4. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

5. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

6. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

7. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;  
descriptor_config.dst_increment_enable = false;  
descriptor_config.block_transfer_count = DATA_LENGTH;  
descriptor_config.source_address = (uint32_t)buffer + DATA_LENGTH;  
descriptor_config.destination_address =  
    (uint32_t)(&i2c_master_instance.hw->I2CM.DATA.reg);
```

8. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

17.8.3.3. Implementation

Code

Add to user application `main()`:

```
dma_start_transfer_job(&example_resource);  
  
i2c_master_dma_set_transfer(&i2c_master_instance, SLAVE_ADDRESS,  
    DATA_LENGTH, I2C_TRANSFER_WRITE);  
  
while (!transfer_is_done) {  
    /* Wait for transfer done */  
}  
  
while (true) {  
}
```

Workflow

1. Start the DMA transfer job.

```
dma_start_transfer_job(&example_resource);
```

2. Set the auto address length and enable flag.

```
i2c_master_dma_set_transfer(&i2c_master_instance, SLAVE_ADDRESS,  
                           DATA_LENGTH, I2C_TRANSFER_WRITE);
```

3. Waiting for transfer complete.

```
while (!transfer_is_done) {  
    /* Wait for transfer done */  
}
```

4. Enter an infinite loop once transfer complete.

```
while (true) {  
}
```

18. SAM I²C Slave Mode (SERCOM I2C) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's SERCOM I²C module, for the transfer of data via an I²C bus. The following driver API modes are covered by this manual:

- Slave Mode Polled APIs
- Slave Mode Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

18.1. Prerequisites

There are no prerequisites.

18.2. Module Overview

The outline of this section is as follows:

- [Driver Feature Macro Definition](#)
- [Functional Description](#)
- [Bus Topology](#)
- [Transactions](#)
- [Multi Master](#)
- [Bus States](#)
- [Bus Timing](#)
- [Operation in Sleep Modes](#)

18.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_I2C_10_BIT_ADDRESS	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_I2C_SCL_STRETCH_MODE	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_I2C_SCL_EXTEND_TIMEOUT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

Note: When using the I2C high-speed mode for off-board communication, there are various high frequency interference, which can lead to distortion of the signals and communication failure. When using Xplained Pro boards in order to test I2C high-speed communication, the following recommendation should be followed:

- Use the SDA-line on PA08 and SCL-line on PA09 for both boards. This will provide stronger pull-ups on both SDA and SCL.
- The SCL should not be higher than 1.5MHz.

18.2.2. Functional Description

The I²C provides a simple two-wire bidirectional bus consisting of a wired-AND type serial clock line (SCL) and a wired-AND type serial data line (SDA).

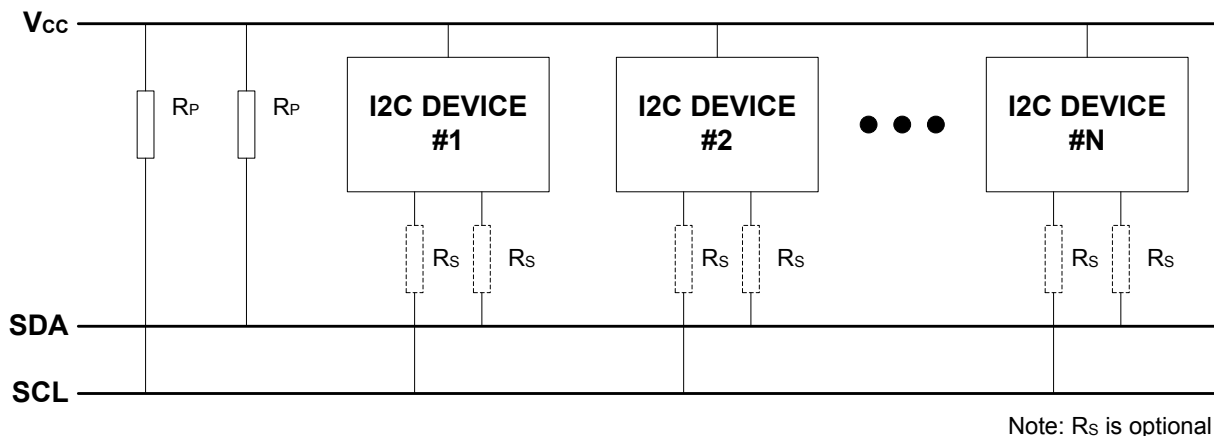
The I²C bus provides a simple, but efficient method of interconnecting multiple master and slave devices. An arbitration mechanism is provided for resolving bus ownership between masters, as only one master device may own the bus at any given time. The arbitration mechanism relies on the wired-AND connections to avoid bus drivers short-circuiting.

A unique address is assigned to all slave devices connected to the bus. A device can contain both master and slave logic, and can emulate multiple slave devices by responding to more than one address.

18.2.3. Bus Topology

The I²C bus topology is illustrated in [Figure 18-1](#). The pull-up resistors (Rs) will provide a high level on the bus lines when none of the I²C devices are driving the bus. These are optional, and can be replaced with a constant current source.

Figure 18-1. I²C Bus Topology



18.2.4. Transactions

The I²C standard defines three fundamental transaction formats:

- Master Write
 - The master transmits data packets to the slave after addressing it
- Master Read
 - The slave transmits data packets to the master after being addressed
- Combined Read/Write
 - A combined transaction consists of several write and read transactions

A data transfer starts with the master issuing a **Start** condition on the bus, followed by the address of the slave together with a bit to indicate whether the master wants to read from or write to the slave. The addressed slave must respond to this by sending an **ACK** back to the master.

After this, data packets are sent from the master or slave, according to the read/write bit. Each packet must be acknowledged (ACK) or not acknowledged (NACK) by the receiver.

If a slave responds with a NACK, the master must assume that the slave cannot receive any more data and cancel the write operation.

The master completes a transaction by issuing a **Stop** condition.

A master can issue multiple **Start** conditions during a transaction; this is then called a **Repeated Start** condition.

18.2.4.1. Address Packets

The slave address consists of seven bits. The 8th bit in the transfer determines the data direction (read or write). An address packet always succeeds a **Start** or **Repeated Start** condition. The 8th bit is handled in the driver, and the user will only have to provide the 7-bit address.

18.2.4.2. Data Packets

Data packets are nine bits long, consisting of one 8-bit data byte, and an acknowledgement bit. Data packets follow either an address packet or another data packet on the bus.

18.2.4.3. Transaction Examples

The gray bits in the following examples are sent from master to slave, and the white bits are sent from slave to master. Example of a read transaction is shown in [Figure 18-2](#). Here, the master first issues a

Start condition and gets ownership of the bus. An address packet with the direction flag set to read is then sent and acknowledged by the slave. Then the slave sends one data packet which is acknowledged by the master. The slave sends another packet, which is not acknowledged by the master and indicates that the master will terminate the transaction. In the end, the transaction is terminated by the master issuing a **Stop** condition.

Figure 18-2. I²C Packet Read

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit 11	Bit 12	Bit 13	Bit 14	Bit 15	Bit 16	Bit 17	Bit 18	Bit 19	Bit 20	Bit 21	Bit 22	Bit 23	Bit 24	Bit 25	Bit 26	Bit 27	Bit 28
START	ADDRESS							READ	ACK	DATA									ACK	DATA							NACK	STOP

Example of a write transaction is shown in [Figure 18-3](#). Here, the master first issues a **Start** condition and gets ownership of the bus. An address packet with the dir flag set to write is then sent and acknowledged by the slave. Then the master sends two data packets, each acknowledged by the slave. In the end, the transaction is terminated by the master issuing a **Stop** condition.

Figure 18-3. I²C Packet Write

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit 11	Bit 12	Bit 13	Bit 14	Bit 15	Bit 16	Bit 17	Bit 18	Bit 19	Bit 20	Bit 21	Bit 22	Bit 23	Bit 24	Bit 25	Bit 26	Bit 27	Bit 28
START	ADDRESS							WRITE	ACK	DATA									ACK	DATA							ACK	STOP

18.2.4.4. Packet Timeout

When a master sends an I²C packet, there is no way of being sure that a slave will acknowledge the packet. To avoid stalling the device forever while waiting for an acknowledge, a user selectable timeout is provided in the `i2c_master_config` struct which lets the driver exit a read or write operation after the specified time. The function will then return the `STATUS_ERR_TIMEOUT` flag.

This is also the case for the slave when using the functions postfixed `_wait`.

The time before the timeout occurs, will be the same as for [unknown bus state](#) timeout.

18.2.4.5. Repeated Start

To issue a **Repeated Start**, the functions postfixed `_no_stop` must be used. These functions will not send a **Stop** condition when the transfer is done, thus the next transfer will start with a **Repeated Start**. To end the transaction, the functions without the `_no_stop` postfix must be used for the last read/write.

18.2.5. Multi Master

In a multi master environment, arbitration of the bus is important, as only one master can own the bus at any point.

18.2.5.1. Arbitration

Clock stretching

The serial clock line is always driven by a master device. However, all devices connected to the bus are allowed stretch the low period of the clock to slow down the overall clock frequency or to insert wait states while processing data. Both master and slave can randomly stretch the clock, which will force the other device into a wait-state until the clock line goes high again.

Arbitration on the data line

If two masters start transmitting at the same time, they will both transmit until one master detects that the other master is pulling the data line low. When this is detected, the master not pulling the line low, will stop the transmission and wait until the bus is idle. As it is the master trying to contact the slave with the lowest address that will get the bus ownership, this will create an arbitration scheme always prioritizing the slaves with the lowest address in case of a bus collision.

18.2.5.2. Clock Synchronization

In situations where more than one master is trying to control the bus clock line at the same time, a clock synchronization algorithm based on the same principles used for clock stretching is necessary.

18.2.6. Bus States

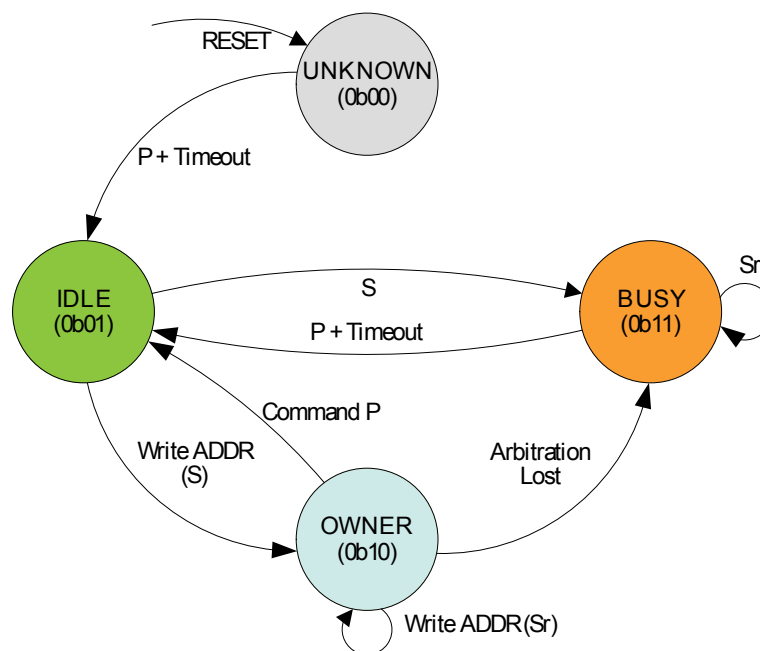
As the I²C bus is limited to one transaction at the time, a master that wants to perform a bus transaction must wait until the bus is free. Because of this, it is necessary for all masters in a multi-master system to know the current status of the bus to be able to avoid conflicts and to ensure data integrity.

- **IDLE** No activity on the bus (between a **Stop** and a new **Start** condition)
- **OWNER** If the master initiates a transaction successfully
- **BUSY** If another master is driving the bus
- **UNKNOWN** If the master has recently been enabled or connected to the bus. Is forced to **IDLE** after given **timeout** when the master module is enabled

The bus state diagram can be seen in [Figure 18-4](#).

- S: Start condition
- P: Stop condition
- Sr: Repeated start condition

Figure 18-4. I²C Bus State Diagram



18.2.7. Bus Timing

Inactive bus timeout for the master and SDA hold time is configurable in the drivers.

18.2.7.1. Unknown Bus State Timeout

When a master is enabled or connected to the bus, the bus state will be unknown until either a given timeout or a stop command has occurred. The timeout is configurable in the `i2c_master_config` struct. The timeout time will depend on toolchain and optimization level used, as the timeout is a loop incrementing a value until it reaches the specified timeout value.

18.2.7.2. SDA Hold Timeout

When using the I²C in slave mode, it will be important to set a SDA hold time which assures that the master will be able to pick up the bit sent from the slave. The SDA hold time makes sure that this is the case by holding the data line low for a given period after the negative edge on the clock.

The SDA hold time is also available for the master driver, but is not a necessity.

18.2.8. Operation in Sleep Modes

The I²C module can operate in all sleep modes by setting the `run_in_standby` Boolean in the `i2c_master_config` or `i2c_slave_config` struct. The operation in slave and master mode is shown in [Table 18-1](#).

Table 18-1. I²C Standby Operations

Run in standby	Slave	Master
false	Disabled, all reception is dropped	Generic Clock (GCLK) disabled when master is idle
true	Wake on address match when enabled	GCLK enabled while in sleep modes

18.3. Special Considerations

18.3.1. Interrupt-driven Operation

While an interrupt-driven operation is in progress, subsequent calls to a write or read operation will return the `STATUS_BUSY` flag, indicating that only one operation is allowed at any given time.

To check if another transmission can be initiated, the user can either call another transfer operation, or use the `i2c_master_get_job_status`/[i2c_slave_get_job_status](#) functions depending on mode.

If the user would like to get callback from operations while using the interrupt-driven driver, the callback must be registered and then enabled using the "register_callback" and "enable_callback" functions.

18.4. Extra Information

For extra information, see [Extra Information for SERCOM I2C Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

18.5. Examples

For a list of examples related to this driver, see [Examples for SERCOM I2C Driver](#).

18.6. API Overview

18.6.1. Structure Definitions

18.6.1.1. Struct `i2c_slave_config`

This is the configuration structure for the I²C slave device. It is used as an argument for `i2c_slave_init` to provide the desired configurations for the module. The structure should be initialized using the `i2c_slave_get_config_defaults`.

Table 18-2. Members

Type	Name	Description
uint16_t	address	Address or upper limit of address range
uint16_t	address_mask	Address mask, second address, or lower limit of address range
enum i2c_slave_address_mode	address_mode	Addressing mode
uint16_t	buffer_timeout	Timeout to wait for master in polled functions
bool	enable_general_call_address	Enable general call address recognition (general call address is defined as 0000000 with direction bit 0).
bool	enable_nack_on_address	Enable NACK on address match (this can be changed after initialization via the i2c_slave_enable_nack_on_address and i2c_slave_disable_nack_on_address functions).
bool	enable_scl_low_timeout	Set to enable the SCL low timeout
enum gclk_generator	generator_source	GCLK generator to use as clock source
uint32_t	pinmux_pad0	PAD0 (SDA) pinmux
uint32_t	pinmux_pad1	PAD1 (SCL) pinmux
bool	run_in_standby	Set to keep module active in sleep modes
bool	scl_low_timeout	Set to enable SCL low time-out
bool	scl_stretch_only_after_ack_bit	Set to enable SCL stretch only after ACK bit (required for high speed)
enum i2c_slave_sda_hold_time	sda_hold_time	SDA hold time with respect to the negative edge of SCL
bool	slave_scl_low_extend_timeout	Set to enable slave SCL low extend time-out

Type	Name	Description
bool	ten_bit_address	Enable 10-bit addressing
enum i2c_slave_transfer_speed	transfer_speed	Transfer speed mode

18.6.1.2. Struct i2c_slave_module

SERCOM I²C slave driver software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

18.6.1.3. Struct i2c_slave_packet

Structure to be used when transferring I²C slave packets.

Table 18-3. Members

Type	Name	Description
uint8_t *	data	Data array containing all data to be transferred
uint16_t	data_length	Length of data array

18.6.2. Macro Definitions

18.6.2.1. Driver Feature Definition

Define SERCOM I²C driver features set according to different device family.

Macro FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED

```
#define FEATURE_I2C_FAST_MODE_PLUS_AND_HIGH_SPEED
```

Fast mode plus and high speed support.

Macro FEATURE_I2C_10_BIT_ADDRESS

```
#define FEATURE_I2C_10_BIT_ADDRESS
```

10-bit address support

Macro FEATURE_I2C_SCL_STRETCH_MODE

```
#define FEATURE_I2C_SCL_STRETCH_MODE
```

SCL stretch mode support

Macro FEATURE_I2C_SCL_EXTEND_TIMEOUT

```
#define FEATURE_I2C_SCL_EXTEND_TIMEOUT
```

SCL extend timeout support

Macro FEATURE_I2C_DMA_SUPPORT

```
#define FEATURE_I2C_DMA_SUPPORT
```

18.6.2.2. I²C Slave Status Flags

I²C slave status flags, returned by `i2c_slave_get_status()` and cleared by `i2c_slave_clear_status()`.

Macro I2C_SLAVE_STATUS_ADDRESS_MATCH

```
#define I2C_SLAVE_STATUS_ADDRESS_MATCH
```

Address Match.

Note: Should only be cleared internally by driver.

Macro I2C_SLAVE_STATUS_DATA_READY

```
#define I2C_SLAVE_STATUS_DATA_READY
```

Data Ready.

Macro I2C_SLAVE_STATUS_STOP_RECEIVED

```
#define I2C_SLAVE_STATUS_STOP_RECEIVED
```

Stop Received.

Macro I2C_SLAVE_STATUS_CLOCK_HOLD

```
#define I2C_SLAVE_STATUS_CLOCK_HOLD
```

Clock Hold.

Note: Cannot be cleared, only valid when I2C_SLAVE_STATUS_ADDRESS_MATCH is set.

Macro I2C_SLAVE_STATUS_SCL_LOW_TIMEOUT

```
#define I2C_SLAVE_STATUS_SCL_LOW_TIMEOUT
```

SCL Low Timeout.

Macro I2C_SLAVE_STATUS_REPEATED_START

```
#define I2C_SLAVE_STATUS_REPEATED_START
```

Repeated Start.

Note: Cannot be cleared, only valid when I2C_SLAVE_STATUS_ADDRESS_MATCH is set.

Macro I2C_SLAVE_STATUS_RECEIVED_NACK

```
#define I2C_SLAVE_STATUS_RECEIVED_NACK
```

Received not acknowledge.

Note: Cannot be cleared.

Macro I2C_SLAVE_STATUS_COLLISION

```
#define I2C_SLAVE_STATUS_COLLISION
```

Transmit Collision.

Macro I2C_SLAVE_STATUS_BUS_ERROR

```
#define I2C_SLAVE_STATUS_BUS_ERROR
```

Bus error.

18.6.3. Function Definitions

18.6.3.1. Lock/Unlock

Function `i2c_slave_lock()`

Attempt to get lock on driver instance.

```
enum status_code i2c_slave_lock(  
    struct i2c_slave_module *const module)
```

This function checks the instance's lock, which indicates whether or not it is currently in use, and sets the lock if it was not already set.

The purpose of this is to enable exclusive access to driver instances, so that, e.g., transactions by different services will not interfere with each other.

Table 18-4. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 18-5. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

Function `i2c_slave_unlock()`

Unlock driver instance.

```
void i2c_slave_unlock(  
    struct i2c_slave_module *const module)
```

This function clears the instance lock, indicating that it is available for use.

Table 18-6. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 18-7. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

18.6.3.2. Configuration and Initialization

Function `i2c_slave_is_syncing()`

Returns the synchronization status of the module.

```
bool i2c_slave_is_syncing(  
    const struct i2c_slave_module *const module)
```

Returns the synchronization status of the module.

Table 18-8. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to software module structure

Returns

Status of the synchronization.

Table 18-9. Return Values

Return value	Description
true	Module is busy synchronizing
false	Module is not synchronizing

Function `i2c_slave_get_config_defaults()`

Gets the I²C slave default configurations.

```
void i2c_slave_get_config_defaults(  
    struct i2c_slave_config *const config)
```

This will initialize the configuration structure to known default values.

The default configuration is as follows:

- Disable SCL low timeout
- 300ns - 600ns SDA hold time
- Buffer timeout = 65535
- Address with mask
- Address = 0
- Address mask = 0 (one single address)
- General call address disabled
- Address nack disabled if the interrupt driver is used
- GCLK generator 0
- Do not run in standby
- PINMUX_DEFAULT for SERCOM pads

Those default configuration only available if the device supports it:

- Not using 10-bit addressing
- Standard-mode and Fast-mode transfer speed
- SCL stretch disabled
- Slave SCL low extend time-out disabled

Table 18-10. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to configuration structure to be initialized

Function i2c_slave_init()

Initializes the requested I²C hardware module.

```
enum status_code i2c_slave_init(
    struct i2c_slave_module *const module,
    Sercom *const hw,
    const struct i2c_slave_config *const config)
```

Initializes the SERCOM I²C slave device requested and sets the provided software module struct. Run this function before any further use of the driver.

Table 18-11. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to software module struct
[in]	hw	Pointer to the hardware instance
[in]	config	Pointer to the configuration struct

Returns

Status of initialization.

Table 18-12. Return Values

Return value	Description
STATUS_OK	Module initiated correctly
STATUS_ERR_DENIED	If module is enabled
STATUS_BUSY	If module is busy resetting
STATUS_ERR_ALREADY_INITIALIZED	If setting other GCLK generator than previously set

Function i2c_slave_enable()

Enables the I²C module.

```
void i2c_slave_enable(
    const struct i2c_slave_module *const module)
```

This will enable the requested I²C module.

Table 18-13. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software module struct

Function i2c_slave_disable()

Disables the I²C module.

```
void i2c_slave_disable(  
    const struct i2c_slave_module *const module)
```

This will disable the I²C module specified in the provided software module structure.

Table 18-14. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software module struct

Function i2c_slave_reset()

Resets the hardware module.

```
void i2c_slave_reset(  
    struct i2c_slave_module *const module)
```

This will reset the module to hardware defaults.

Table 18-15. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

18.6.3.3. Read and Write

Function i2c_slave_write_packet_wait()

Writes a packet to the master.

```
enum status_code i2c_slave_write_packet_wait(  
    struct i2c_slave_module *const module,  
    struct i2c_slave_packet *const packet)
```

Writes a packet to the master. This will wait for the master to issue a request.

Table 18-16. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure
[in]	packet	Packet to write to master

Returns

Status of packet write.

Table 18-17. Return Values

Return value	Description
STATUS_OK	Packet was written successfully
STATUS_ERR_DENIED	Start condition not received, another interrupt flag is set

Return value	Description
STATUS_ERR_IO	There was an error in the previous transfer
STATUS_ERR_BAD_FORMAT	Master wants to write data
STATUS_ERR_INVALID_ARG	Invalid argument(s) was provided
STATUS_ERR_BUSY	The I ² C module is busy with a job
STATUS_ERR_ERR_OVERFLOW	Master NACKed before entire packet was transferred
STATUS_ERR_TIMEOUT	No response was given within the timeout period

Function `i2c_slave_read_packet_wait()`

Reads a packet from the master.

```
enum status_code i2c_slave_read_packet_wait(
    struct i2c_slave_module *const module,
    struct i2c_slave_packet *const packet)
```

Reads a packet from the master. This will wait for the master to issue a request.

Table 18-18. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure
[out]	packet	Packet to read from master

Returns

Status of packet read.

Table 18-19. Return Values

Return value	Description
STATUS_OK	Packet was read successfully
STATUS_ABORTED	Master sent stop condition or repeated start before specified length of bytes was received
STATUS_ERR_IO	There was an error in the previous transfer
STATUS_ERR_DENIED	Start condition not received, another interrupt flag is set
STATUS_ERR_INVALID_ARG	Invalid argument(s) was provided
STATUS_ERR_BUSY	The I ² C module is busy with a job
STATUS_ERR_BAD_FORMAT	Master wants to read data
STATUS_ERR_ERR_OVERFLOW	Last byte received overflows buffer

Function `i2c_slave_get_direction_wait()`

Waits for a start condition on the bus.

```
enum i2c_slave_direction i2c_slave_get_direction_wait(  
    struct i2c_slave_module *const module)
```

Note: This function is only available for 7-bit slave addressing.

Waits for the master to issue a start condition on the bus.

Note: This function does not check for errors in the last transfer, this will be discovered when reading or writing.

Table 18-20. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure

Returns

Direction of the current transfer, when in slave mode.

Table 18-21. Return Values

Return value	Description
I2C_SLAVE_DIRECTION_NONE	No request from master within timeout period
I2C_SLAVE_DIRECTION_READ	Write request from master
I2C_SLAVE_DIRECTION_WRITE	Read request from master

18.6.3.4. Status Management

Function `i2c_slave_get_status()`

Retrieves the current module status.

```
uint32_t i2c_slave_get_status(  
    struct i2c_slave_module *const module)
```

Checks the status of the module and returns it as a bitmask of status flags.

Table 18-22. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the I ² C slave software device struct

Returns

Bitmask of status flags.

Table 18-23. Return Values

Return value	Description
I2C_SLAVE_STATUS_ADDRESS_MATCH	A valid address has been received
I2C_SLAVE_STATUS_DATA_READY	A I ² C slave byte transmission is successfully completed

Return value	Description
I2C_SLAVE_STATUS_STOP_RECEIVED	A stop condition is detected for a transaction being processed
I2C_SLAVE_STATUS_CLOCK_HOLD	The slave is holding the SCL line low
I2C_SLAVE_STATUS_SCL_LOW_TIMEOUT	An SCL low time-out has occurred
I2C_SLAVE_STATUS_REPEATED_START	Indicates a repeated start, only valid if I2C_SLAVE_STATUS_ADDRESS_MATCH is set
I2C_SLAVE_STATUS_RECEIVED_NACK	The last data packet sent was not acknowledged
I2C_SLAVE_STATUS_COLLISION	The I ² C slave was not able to transmit a high data or NACK bit
I2C_SLAVE_STATUS_BUS_ERROR	An illegal bus condition has occurred on the bus

Function `i2c_slave_clear_status()`

Clears a module status flag.

```
void i2c_slave_clear_status(
    struct i2c_slave_module *const module,
    uint32_t status_flags)
```

Clears the given status flag of the module.

Note: Not all status flags can be cleared.

Table 18-24. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the I ² C software device struct
[in]	status_flags	Bit mask of status flags to clear

18.6.3.5. SERCOM I²C slave with DMA Interfaces

Function `i2c_slave_dma_read_interrupt_status()`

Read SERCOM I²C interrupt status.

```
uint8_t i2c_slave_dma_read_interrupt_status(
    struct i2c_slave_module *const module)
```

Read I²C interrupt status for DMA transfer.

Table 18-25. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Function `i2c_slave_dma_write_interrupt_status()`

Write SERCOM I²C interrupt status.

```
void i2c_slave_dma_write_interrupt_status(  
    struct i2c_slave_module *const module,  
    uint8_t flag)
```

Write I²C interrupt status for DMA transfer.

Table 18-26. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock
[in]	flag	Interrupt flag status

18.6.3.6. Address Match Functionality

Function `i2c_slave_enable_nack_on_address()`

Enables sending of NACK on address match.

```
void i2c_slave_enable_nack_on_address(  
    struct i2c_slave_module *const module)
```

Enables sending of NACK on address match, thus discarding any incoming transaction.

Table 18-27. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

Function `i2c_slave_disable_nack_on_address()`

Disables sending NACK on address match.

```
void i2c_slave_disable_nack_on_address(  
    struct i2c_slave_module *const module)
```

Disables sending of NACK on address match, thus acknowledging incoming transactions.

Table 18-28. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

18.6.3.7. Callbacks

Function `i2c_slave_register_callback()`

Registers callback for the specified callback type.

```
void i2c_slave_register_callback(  
    struct i2c_slave_module *const module,  
    i2c_slave_callback_t callback,  
    enum i2c_slave_callback callback_type)
```

Associates the given callback function with the specified callback type. To enable the callback, the `i2c_slave_enable_callback` function must be used.

Table 18-29. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback	Pointer to the function desired for the specified callback
[in]	callback_type	Callback type to register

Function i2c_slave_unregister_callback()

Unregisters callback for the specified callback type.

```
void i2c_slave_unregister_callback(
    struct i2c_slave_module *const module,
    enum i2c_slave_callback callback_type)
```

Removes the currently registered callback for the given callback type.

Table 18-30. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Callback type to unregister

Function i2c_slave_enable_callback()

Enables callback.

```
void i2c_slave_enable_callback(
    struct i2c_slave_module *const module,
    enum i2c_slave_callback callback_type)
```

Enables the callback specified by the callback_type.

Table 18-31. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Callback type to enable

Function i2c_slave_disable_callback()

Disables callback.

```
void i2c_slave_disable_callback(
    struct i2c_slave_module *const module,
    enum i2c_slave_callback callback_type)
```

Disables the callback specified by the callback_type.

Table 18-32. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software module struct
[in]	callback_type	Callback type to disable

18.6.3.8. Read and Write, Interrupt-Driven

Function `i2c_slave_read_packet_job()`

Initiates a reads packet operation.

```
enum status_code i2c_slave_read_packet_job(
    struct i2c_slave_module *const module,
    struct i2c_slave_packet *const packet)
```

Reads a data packet from the master. A write request must be initiated by the master before the packet can be read.

The `I2C_SLAVE_CALLBACK_WRITE_REQUEST` callback can be used to call this function.

Table 18-33. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting asynchronously reading I²C packet.

Table 18-34. Return Values

Return value	Description
STATUS_OK	If reading was started successfully
STATUS_BUSY	If module is currently busy with another transfer

Function `i2c_slave_write_packet_job()`

Initiates a write packet operation.

```
enum status_code i2c_slave_write_packet_job(
    struct i2c_slave_module *const module,
    struct i2c_slave_packet *const packet)
```

Writes a data packet to the master. A read request must be initiated by the master before the packet can be written.

The `I2C_SLAVE_CALLBACK_READ_REQUEST` callback can be used to call this function.

Table 18-35. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module struct
[in, out]	packet	Pointer to I ² C packet to transfer

Returns

Status of starting writing I²C packet.

Table 18-36. Return Values

Return value	Description
STATUS_OK	If writing was started successfully
STATUS_BUSY	If module is currently busy with another transfer

Function i2c_slave_cancel_job()

Cancels any currently ongoing operation.

```
void i2c_slave_cancel_job(
    struct i2c_slave_module *const module)
```

Terminates the running transfer operation.

Table 18-37. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

Function i2c_slave_get_job_status()

Gets status of ongoing job.

```
enum status_code i2c_slave_get_job_status(
    struct i2c_slave_module *const module)
```

Will return the status of the ongoing job, or the error that occurred in the last transfer operation. The status will be cleared when starting a new job.

Table 18-38. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to software module structure

Returns

Status of job.

Table 18-39. Return Values

Return value	Description
STATUS_OK	No error has occurred
STATUS_BUSY	Transfer is in progress
STATUS_ERR_IO	A collision, timeout or bus error happened in the last transfer
STATUS_ERR_TIMEOUT	A timeout occurred
STATUS_ERR_OVERFLOW	Data from master overflows receive buffer

18.6.4. Enumeration Definitions

18.6.4.1. Enum i2c_slave_address_mode

Enum for the possible address modes.

Table 18-40. Members

Enum value	Description
I2C_SLAVE_ADDRESS_MODE_MASK	Address match on address_mask used as a mask to address
I2C_SLAVE_ADDRESS_MODE_TWO_ADDRESSES	Address math on both address and address_mask
I2C_SLAVE_ADDRESS_MODE_RANGE	Address match on range of addresses between and including address and address_mask

18.6.4.2. Enum i2c_slave_callback

The available callback types for the I²C slave.

Table 18-41. Members

Enum value	Description
I2C_SLAVE_CALLBACK_WRITE_COMPLETE	Callback for packet write complete
I2C_SLAVE_CALLBACK_READ_COMPLETE	Callback for packet read complete
I2C_SLAVE_CALLBACK_READ_REQUEST	Callback for read request from master - can be used to issue a write
I2C_SLAVE_CALLBACK_WRITE_REQUEST	Callback for write request from master - can be used to issue a read
I2C_SLAVE_CALLBACK_ERROR	Callback for error
I2C_SLAVE_CALLBACK_ERROR_LAST_TRANSFER	Callback for error in last transfer. Discovered on a new address interrupt.

18.6.4.3. Enum i2c_slave_direction

Enum for the direction of a request.

Table 18-42. Members

Enum value	Description
I2C_SLAVE_DIRECTION_READ	Read
I2C_SLAVE_DIRECTION_WRITE	Write
I2C_SLAVE_DIRECTION_NONE	No direction

18.6.4.4. Enum i2c_slave_sda_hold_time

Enum for the possible SDA hold times with respect to the negative edge of SCL.

Table 18-43. Members

Enum value	Description
I2C_SLAVE_SDA_HOLD_TIME_DISABLED	SDA hold time disabled
I2C_SLAVE_SDA_HOLD_TIME_50NS_100NS	SDA hold time 50ns - 100ns
I2C_SLAVE_SDA_HOLD_TIME_300NS_600NS	SDA hold time 300ns - 600ns
I2C_SLAVE_SDA_HOLD_TIME_400NS_800NS	SDA hold time 400ns - 800ns

18.6.4.5. Enum i2c_slave_transfer_speed

Enum for the transfer speed.

Table 18-44. Members

Enum value	Description
I2C_SLAVE_SPEED_STANDARD_AND_FAST	Standard-mode (Sm) up to 100KHz and Fast-mode (Fm) up to 400KHz
I2C_SLAVE_SPEED_FAST_MODE_PLUS	Fast-mode Plus (Fm+) up to 1MHz
I2C_SLAVE_SPEED_HIGH_SPEED	High-speed mode (Hs-mode) up to 3.4MHz

18.6.4.6. Enum i2c_transfer_direction

For master: transfer direction or setting direction bit in address. For slave: direction of request from master.

Table 18-45. Members

Enum value	Description
I2C_TRANSFER_WRITE	Master write operation is in progress
I2C_TRANSFER_READ	Master read operation is in progress

18.7. Extra Information for SERCOM I²C Driver

18.7.1. Acronyms

[Table 18-46](#) is a table listing the acronyms used in this module, along with their intended meanings.

Table 18-46. Acronyms

Acronym	Description
SDA	Serial Data Line
SCL	Serial Clock Line
SERCOM	Serial Communication Interface
DMA	Direct Memory Access

18.7.2. Dependencies

The I²C driver has the following dependencies:

- System Pin Multiplexer Driver

18.7.3. Errata

There are no errata related to this driver.

18.7.4. Module History

[Table 18-47](#) is an overview of the module history, detailing enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version listed in [Table 18-47](#).

Table 18-47. Module History

Changelog
• Added 10-bit addressing and high speed support in SAM D21 • Separate structure i2c_packet into i2c_master_packet and i2c_slave packet
• Added support for SCL stretch and extended timeout hardware features in SAM D21 • Added fast mode plus support in SAM D21
Fixed incorrect logical mask for determining if a bus error has occurred in I ² C Slave mode
Initial Release

18.8. Examples for SERCOM I²C Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM I2C Slave Mode \(SERCOM I2C\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for the I2C Slave module - Basic Use Case](#)

- [Quick Start Guide for the I2C Slave module - Callback Use Case](#)
- [Quick Start Guide for the I2C Slave module - DMA Use Case](#)

18.8.1. Quick Start Guide for SERCOM I²C Slave - Basic

In this use case, the I²C will be used and set up as follows:

- Slave mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value

18.8.1.1. Prerequisites

The device must be connected to an I²C master.

18.8.1.2. Setup

Code

The following must be added to the user application:

A sample buffer to write from, a sample buffer to read to and length of buffers:

```
#define DATA_LENGTH 10

uint8_t write_buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09
};
uint8_t read_buffer[DATA_LENGTH];
```

Address to respond to:

```
#define SLAVE_ADDRESS 0x12
```

Globally accessible module structure:

```
struct i2c_slave_module i2c_slave_instance;
```

Function for setting up the module:

```
void configure_i2c_slave(void)
{
    /* Create and initialize config_i2c_slave structure */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);

    /* Change address and address_mode */
    config_i2c_slave.address = SLAVE_ADDRESS;
    config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
#ifdef SAMR30
    config_i2c_slave.pinmux_pad0 = CONF_SLAVE_SDA_PINMUX;
    config_i2c_slave.pinmux_pad1 = CONF_SLAVE_SCK_PINMUX;
#endif
    config_i2c_slave.buffer_timeout = 1000;

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
        &config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

Add to user application `main()`:

```
configure_i2c_slave();

enum i2c_slave_direction dir;
struct i2c_slave_packet packet = {
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
};
```

Workflow

1. Configure and enable module.

```
configure_i2c_slave();
```

1. Create and initialize configuration structure.

```
struct i2c_slave_config config_i2c_slave;
i2c_slave_get_config_defaults(&config_i2c_slave);
```

2. Change address and address mode settings in the configuration.

```
config_i2c_slave.address      = SLAVE_ADDRESS;
config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
config_i2c_slave.buffer_timeout = 1000;
```

3. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);
```

4. Enable the module.

```
i2c_slave_enable(&i2c_slave_instance);
```

2. Create variable to hold transfer direction.

```
enum i2c_slave_direction dir;
```

3. Create packet variable to transfer.

```
struct i2c_slave_packet packet = {
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
};
```

18.8.1.3. Implementation

Code

Add to user application `main()`:

```
while (true) {
    /* Wait for direction from master */
    dir = i2c_slave_get_direction_wait(&i2c_slave_instance);

    /* Transfer packet in direction requested by master */
    if (dir == I2C_SLAVE_DIRECTION_READ) {
        packet.data = read_buffer;
        i2c_slave_read_packet_wait(&i2c_slave_instance, &packet);
    } else if (dir == I2C_SLAVE_DIRECTION_WRITE) {
        packet.data = write_buffer;
        i2c_slave_write_packet_wait(&i2c_slave_instance, &packet);
    }
}
```

Workflow

1. Wait for start condition from master and get transfer direction.

```
dir = i2c_slave_get_direction_wait(&i2c_slave_instance);
```

2. Depending on transfer direction, set up buffer to read to or write from, and write or read from master.

```
if (dir == I2C_SLAVE_DIRECTION_READ) {
    packet.data = read_buffer;
    i2c_slave_read_packet_wait(&i2c_slave_instance, &packet);
} else if (dir == I2C_SLAVE_DIRECTION_WRITE) {
    packet.data = write_buffer;
    i2c_slave_write_packet_wait(&i2c_slave_instance, &packet);
}
```

18.8.2. Quick Start Guide for SERCOM I²C Slave - Callback

In this use case, the I²C will be used and set up as follows:

- Slave mode
- 100KHz operation speed
- Not operational in standby
- 10000 packet timeout value

18.8.2.1. Prerequisites

The device must be connected to an I²C master.

18.8.2.2. Setup

Code

The following must be added to the user application:

A sample buffer to write from, a sample buffer to read to and length of buffers:

```
#define DATA_LENGTH 10
static uint8_t write_buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
};
static uint8_t read_buffer [DATA_LENGTH];
```

Address to respond to:

```
#define SLAVE_ADDRESS 0x12
```

Globally accessible module structure:

```
struct i2c_slave_module i2c_slave_instance;
```

Globally accessible packet:

```
static struct i2c_slave_packet packet;
```

Function for setting up the module:

```
void configure_i2c_slave(void)
{
    /* Initialize config structure and module instance */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);
    /* Change address and address_mode */
}
```

```

    config_i2c_slave.address      = SLAVE_ADDRESS;
    config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
#if SAMR30
    config_i2c_slave.pinmux_pad0   = CONF_SLAVE_SDA_PINMUX;
    config_i2c_slave.pinmux_pad1   = CONF_SLAVE_SCK_PINMUX;
#endif
    /* Initialize and enable device with config */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}

```

Callback function for read request from a master:

```

void i2c_read_request_callback(
    struct i2c_slave_module *const module)
{
    /* Init i2c packet */
    packet.data_length = DATA_LENGTH;
    packet.data        = write_buffer;

    /* Write buffer to master */
    i2c_slave_write_packet_job(module, &packet);
}

```

Callback function for write request from a master:

```

void i2c_write_request_callback(
    struct i2c_slave_module *const module)
{
    /* Init i2c packet */
    packet.data_length = DATA_LENGTH;
    packet.data        = read_buffer;

    /* Read buffer from master */
    if (i2c_slave_read_packet_job(module, &packet) != STATUS_OK) {
    }
}

```

Function for setting up the callback functionality of the driver:

```

void configure_i2c_slave_callbacks(void)
{
    /* Register and enable callback functions */
    i2c_slave_register_callback(&i2c_slave_instance,
i2c_read_request_callback,
        I2C_SLAVE_CALLBACK_READ_REQUEST);
    i2c_slave_enable_callback(&i2c_slave_instance,
        I2C_SLAVE_CALLBACK_READ_REQUEST);

    i2c_slave_register_callback(&i2c_slave_instance,
i2c_write_request_callback,
        I2C_SLAVE_CALLBACK_WRITE_REQUEST);
    i2c_slave_enable_callback(&i2c_slave_instance,
        I2C_SLAVE_CALLBACK_WRITE_REQUEST);
}

```

Add to user application main():

```

/* Configure device and enable */
configure_i2c_slave();
configure_i2c_slave_callbacks();

```

Workflow

1. Configure and enable module.

```
configure_i2c_slave();
```

1. Create and initialize configuration structure.

```
struct i2c_slave_config config_i2c_slave;  
i2c_slave_get_config_defaults(&config_i2c_slave);
```

2. Change address and address mode settings in the configuration.

```
config_i2c_slave.address      = SLAVE_ADDRESS;  
config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
```

3. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,  
&config_i2c_slave);
```

4. Enable the module.

```
i2c_slave_enable(&i2c_slave_instance);
```

2. Register and enable callback functions.

```
configure_i2c_slave_callbacks();
```

1. Register and enable callbacks for read and write requests from master.

```
i2c_slave_register_callback(&i2c_slave_instance,  
i2c_read_request_callback,  
I2C_SLAVE_CALLBACK_READ_REQUEST);  
i2c_slave_enable_callback(&i2c_slave_instance,  
I2C_SLAVE_CALLBACK_READ_REQUEST);  
  
i2c_slave_register_callback(&i2c_slave_instance,  
i2c_write_request_callback,  
I2C_SLAVE_CALLBACK_WRITE_REQUEST);  
i2c_slave_enable_callback(&i2c_slave_instance,  
I2C_SLAVE_CALLBACK_WRITE_REQUEST);
```

18.8.2.3. Implementation

Code

Add to user application `main()`:

```
while (true) {  
    /* Infinite loop while waiting for I2C master interaction */  
}
```

Workflow

1. Infinite while loop, while waiting for interaction from master.

```
while (true) {  
    /* Infinite loop while waiting for I2C master interaction */  
}
```

18.8.2.4. Callback

When an address packet is received, one of the callback functions will be called, depending on the DIR bit in the received packet.

Workflow

- Read request callback:
 1. Length of buffer and buffer to be sent to master is initialized.

```
packet.data_length = DATA_LENGTH;  
packet.data        = write_buffer;
```

2. Write packet to master.

```
i2c_slave_write_packet_job(module, &packet);
```

- Write request callback:
 1. Length of buffer and buffer to be read from master is initialized.

```
packet.data_length = DATA_LENGTH;  
packet.data        = read_buffer;
```

2. Read packet from master.

```
if (i2c_slave_read_packet_job(module, &packet) != STATUS_OK) {  
}
```

18.8.3. Quick Start Guide for Using DMA with SERCOM I²C Slave

The supported board list:

- SAMD21 Xplained Pro
- SAMR21 Xplained Pro
- SAML21 Xplained Pro
- SAML22 Xplained Pro
- SAMDA1 Xplained Pro
- SAMC21 Xplained Pro

In this use case, the I²C will be used and set up as follows:

- Slave mode
- 100KHz operation speed
- Not operational in standby
- 65535 unknown bus state timeout value

18.8.3.1. Prerequisites

The device must be connected to an I²C slave.

18.8.3.2. Setup

Code

The following must be added to the user application:

- Address to respond to:

```
#define SLAVE_ADDRESS 0x12
```

- A sample buffer to send, number of entries to send and address of slave:

```
#define DATA_LENGTH 10  
uint8_t read_buffer[DATA_LENGTH];
```

- Globally accessible module structure:

```
struct i2c_slave_module i2c_slave_instance;
```


- Function for setting up the module:

```
void configure_i2c_slave(void)
{
    /* Create and initialize config_i2c_slave structure */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);

    /* Change address and address_mode */
    config_i2c_slave.address      = SLAVE_ADDRESS;
    config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
    config_i2c_slave.buffer_timeout = 1000;
#ifdef SAMR30
    config_i2c_slave.pinmux_pad0 = CONF_SLAVE_SDA_PINMUX;
    config_i2c_slave.pinmux_pad1 = CONF_SLAVE_SCK_PINMUX;
#endif

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

- Globally accessible DMA module structure:

```
struct dma_resource i2c_dma_resource;
```

- Globally accessible DMA transfer descriptor:

```
COMPILER_ALIGNED(16)
DmacDescriptor i2c_dma_descriptor SECTION_DMAC_DESCRIPTOR;
```

- Function for setting up the DMA resource:

```
void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.peripheral_trigger = CONF_I2C_DMA_TRIGGER;
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(resource, &config);
}
```

- Function for setting up the DMA transfer descriptor:

```
void setup_dma_descriptor(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    descriptor_config.src_increment_enable = false;
    descriptor_config.block_transfer_count = DATA_LENGTH;
    descriptor_config.destination_address = (uint32_t)read_buffer +
DATA_LENGTH;
    descriptor_config.source_address =
        (uint32_t)(&i2c_slave_instance.hw->I2CS.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}
```

- Add to user application `main()`:

```
configure_i2c_slave();

configure_dma_resource(&i2c_dma_resource);
setup_dma_descriptor(&i2c_dma_descriptor);
dma_add_descriptor(&i2c_dma_resource, &i2c_dma_descriptor);
```

Workflow

1. Configure and enable module:

```
void configure_i2c_slave(void)
{
    /* Create and initialize config_i2c_slave structure */
    struct i2c_slave_config config_i2c_slave;
    i2c_slave_get_config_defaults(&config_i2c_slave);

    /* Change address and address_mode */
    config_i2c_slave.address      = SLAVE_ADDRESS;
    config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
    config_i2c_slave.buffer_timeout = 1000;
#ifdef SAMR30
    config_i2c_slave.pinmux_pad0 = CONF_SLAVE_SDA_PINMUX;
    config_i2c_slave.pinmux_pad1 = CONF_SLAVE_SCK_PINMUX;
#endif

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

1. Create and initialize configuration structure.

```
struct i2c_slave_config config_i2c_slave;
i2c_slave_get_config_defaults(&config_i2c_slave);
```

2. Change settings in the configuration.

```
config_i2c_slave.address      = SLAVE_ADDRESS;
config_i2c_slave.address_mode = I2C_SLAVE_ADDRESS_MODE_MASK;
config_i2c_slave.buffer_timeout = 1000;
#ifdef SAMR30
    config_i2c_slave.pinmux_pad0 = CONF_SLAVE_SDA_PINMUX;
    config_i2c_slave.pinmux_pad1 = CONF_SLAVE_SCK_PINMUX;
#endif
```

3. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, CONF_I2C_SLAVE_MODULE,
&config_i2c_slave);
```

4. Enable the module.

```
i2c_slave_enable(&i2c_slave_instance);
```

2. Configure DMA

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

2. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM RX trigger causes a beat transfer in this example.

```
config.peripheral_trigger = CONF_I2C_DMA_TRIGGER;  
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

4. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

5. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

6. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

7. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;  
descriptor_config.src_increment_enable = false;  
descriptor_config.block_transfer_count = DATA_LENGTH;  
descriptor_config.destination_address = (uint32_t)read_buffer +  
DATA_LENGTH;  
descriptor_config.source_address =  
    (uint32_t)(&i2c_slave_instance.hw->I2CS.DATA.reg);
```

8. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

18.8.3.3. Implementation

Code

Add to user application `main()`:

```
dma_start_transfer_job(&i2c_dma_resource);  
  
while (true) {  
    if (i2c_slave_dma_read_interrupt_status(&i2c_slave_instance) &  
        SERCOM_I2CS_INTFLAG_AMATCH) {  
        i2c_slave_dma_write_interrupt_status(&i2c_slave_instance,  
            SERCOM_I2CS_INTFLAG_AMATCH);  
    }  
}
```

Workflow

1. Start to wait a packet from master.

```
dma_start_transfer_job(&i2c_dma_resource);
```

2. Once data ready, clear the address match status.

```
while (true) {  
    if (i2c_slave_dma_read_interrupt_status(&i2c_slave_instance) &  
        SERCOM_I2CS_INTFLAG_AMATCH) {  
        i2c_slave_dma_write_interrupt_status(&i2c_slave_instance,  
            SERCOM_I2CS_INTFLAG_AMATCH);  
    }  
}
```

19. SAM Serial Peripheral Interface (SERCOM SPI) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the SERCOM module in its SPI mode to transfer SPI data frames. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

19.1. Prerequisites

There are no prerequisites.

19.2. Module Overview

The Serial Peripheral Interface (SPI) is a high-speed synchronous data transfer interface using three or four pins. It allows fast communication between a master device and one or more peripheral devices.

A device connected to the bus must act as a master or a slave. The master initiates and controls all data transactions. The SPI master initiates a communication cycle by pulling low the Slave Select (SS) pin of the desired slave. The Slave Select pin is active low. Master and slave prepare data to be sent in their respective shift registers, and the master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from master to slave on the Master Out - Slave In (MOSI) line, and from slave to master on the Master In - Slave Out (MISO) line. After each data transfer, the master can synchronize to the slave by pulling the SS line high.

19.2.1. Driver Feature Macro Definition

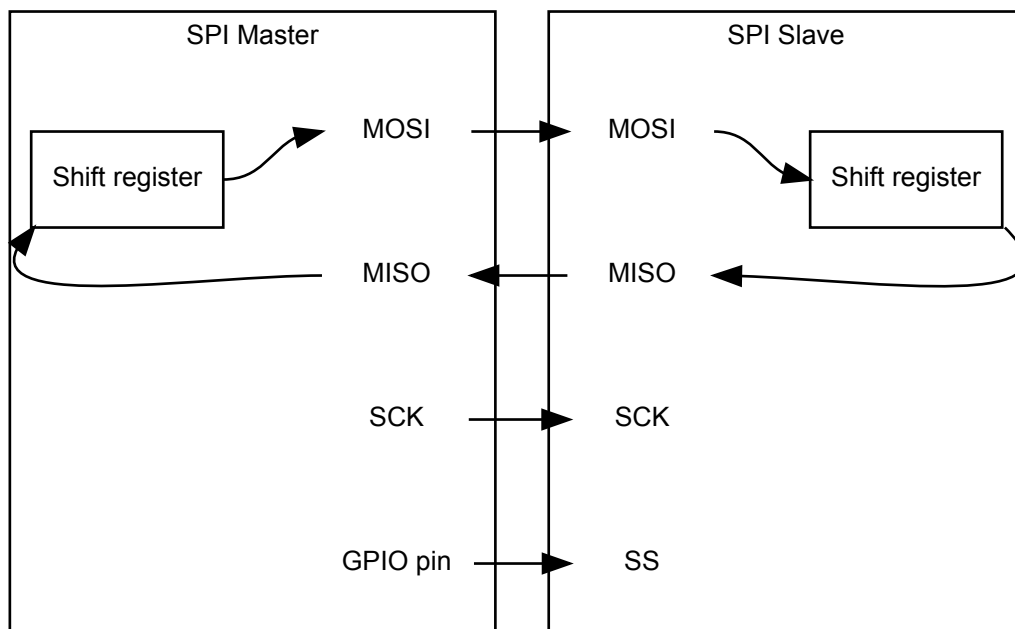
Driver feature macro	Supported devices
FEATURE_SPI_SLAVE_SELECT_LOW_DETECT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_SPI_HARDWARE_SLAVE_SELECT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_SPI_ERROR_INTERRUPT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_SPI_SYNC_SCHEME_VERSION_2	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

19.2.2. SPI Bus Connection

In [Figure 19-1](#), the connection between one master and one slave is shown.

Figure 19-1. SPI Bus Connection



The different lines are as follows:

- **MISO** Master Input Slave Output. The line where the data is shifted out from the slave and into the master.
- **MOSI** Master Output Slave Input. The line where the data is shifted out from the master and into the slave.
- **SCK** Serial Clock. Generated by the master device.
- **SS** Slave Select. To initiate a transaction, the master must pull this line low.

If the bus consists of several SPI slaves, they can be connected in parallel and the SPI master can use general I/O pins to control separate SS lines to each slave on the bus.

It is also possible to connect all slaves in series. In this configuration, a common SS is provided to N slaves, enabling them simultaneously. The MISO from the $N-1$ slaves is connected to the MOSI on the next slave. The N th slave connects its MISO back to the master. For a complete transaction, the master must shift $N+1$ characters.

19.2.3. SPI Character Size

The SPI character size is configurable to eight or nine bits.

19.2.4. Master Mode

When configured as a master, the SS pin will be configured as an output.

19.2.4.1. Data Transfer

Writing a character will start the SPI clock generator, and the character is transferred to the shift register when the shift register is empty. Once this is done, a new character can be written. As each character is shifted out from the master, a character is shifted in from the slave. If the receiver is enabled, the data is moved to the receive buffer at the completion of the frame and can be read.

19.2.5. Slave Mode

When configured as a slave, the SPI interface will remain inactive with MISO tri-stated as long as the SS pin is driven high.

19.2.5.1. Data Transfer

The data register can be updated at any time. As the SPI slave shift register is clocked by SCK, a minimum of three SCK cycles are needed from the time new data is written, until the character is ready to be shifted out. If the shift register has not been loaded with data, the current contents will be transmitted.

If constant transmission of data is needed in SPI slave mode, the system clock should be faster than SCK. If the receiver is enabled, the received character can be read from the receive buffer. When SS line is driven high, the slave will not receive any additional data.

19.2.5.2. Address Recognition

When the SPI slave is configured with address recognition, the first character in a transaction is checked for an address match. If there is a match, the MISO output is enabled and the transaction is processed. If the address does not match, the complete transaction is ignored.

If the device is asleep, it can be woken up by an address match in order to process the transaction.

Note: In master mode, an address packet is written by the `spi_select_slave` function if the `address_enabled` configuration is set in the `spi_slave_inst_config` struct.

19.2.6. Data Modes

There are four combinations of SCK phase and polarity with respect to serial data. Table 19-1 shows the clock polarity (CPOL) and clock phase (CPHA) in the different modes. *Leading edge* is the first clock edge in a clock cycle and *trailing edge* is the last clock edge in a clock cycle.

Table 19-1. SPI Data Modes

Mode	CPOL	CPHA	Leading Edge	Trailing Edge
0	0	0	Rising, Sample	Falling, Setup
1	0	1	Rising, Setup	Falling, Sample
2	1	0	Falling, Sample	Rising, Setup
3	1	1	Falling, Setup	Rising, Sample

19.2.7. SERCOM Pads

The SERCOM pads are automatically configured as seen in [Table 19-2](#). If the receiver is disabled, the data input (MISO for master, MOSI for slave) can be used for other purposes.

In master mode, the SS pin(s) must be configured using the [spi_slave_inst](#) struct.

Table 19-2. SERCOM SPI Pad Usages

Pin	Master SPI	Slave SPI
MOSI	Output	Input
MISO	Input	Output
SCK	Output	Input
SS	User defined output enable	Input

19.2.8. Operation in Sleep Modes

The SPI module can operate in all sleep modes by setting the `run_in_standby` option in the [spi_config](#) struct. The operation in slave and master mode is shown in the table below.

<code>run_in_standby</code>	Slave	Master
false	Disabled, all reception is dropped	GCLK is disabled when master is idle, wake on transmit complete
true	Wake on reception	GCLK is enabled while in sleep modes, wake on all interrupts

19.2.9. Clock Generation

In SPI master mode, the clock (SCK) is generated internally using the SERCOM baudrate generator. In SPI slave mode, the clock is provided by an external master on the SCK pin. This clock is used to directly clock the SPI shift register.

19.3. Special Considerations**19.3.1. pinmux Settings**

The pin MUX settings must be configured properly, as not all settings can be used in different modes of operation.

19.4. Extra Information

For extra information, see [Extra Information for SERCOM SPI Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Workarounds Implemented by Driver](#)
- [Module History](#)

19.5. Examples

For a list of examples related to this driver, see [Examples for SERCOM SPI Driver](#).

19.6. API Overview

19.6.1. Variable and Type Definitions

19.6.1.1. Type spi_callback_t

```
typedef void(* spi_callback_t )(struct spi_module *const module)
```

Type of the callback functions

19.6.2. Structure Definitions

19.6.2.1. Struct spi_config

Configuration structure for an SPI instance. This structure should be initialized by the [spi_get_config_defaults](#) function before being modified by the user application.

Table 19-3. Members

Type	Name	Description
enum spi_character_size	character_size	SPI character size
enum spi_data_order	data_order	Data order
enum gclk_generator	generator_source	GCLK generator to use as clock source
bool	master_slave_select_enable	Enable Master Slave Select
enum spi_mode	mode	SPI mode
union spi_config.mode_specific	mode_specific	Union for slave or master specific configuration
enum spi_signal_mux_setting	mux_setting	MUX setting
uint32_t	pinmux_pad0	PAD0 pinmux

Type	Name	Description
uint32_t	pinmux_pad1	PAD1 pinmux
uint32_t	pinmux_pad2	PAD2 pinmux
uint32_t	pinmux_pad3	PAD3 pinmux
bool	receiver_enable	Enable receiver
bool	run_in_standby	Enabled in sleep modes
bool	select_slave_low_detect_enable	Enable Slave Select Low Detect
enum spi_transfer_mode	transfer_mode	Transfer mode

19.6.2.2. Union `spi_config.mode_specific`

Union for slave or master specific configuration

Table 19-4. Members

Type	Name	Description
struct spi_master_config	master	Master specific configuration
struct spi_slave_config	slave	Slave specific configuration

19.6.2.3. Struct `spi_master_config`

SPI Master configuration structure.

Table 19-5. Members

Type	Name	Description
uint32_t	baudrate	Baud rate

19.6.2.4. Struct `spi_module`

SERCOM SPI driver software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

19.6.2.5. Struct `spi_slave_config`

SPI slave configuration structure.

Table 19-6. Members

Type	Name	Description
uint8_t	address	Address
uint8_t	address_mask	Address mask
enum spi_addr_mode	address_mode	Address mode

Type	Name	Description
enum spi_frame_format	frame_format	Frame format
bool	preload_enable	Preload data to the shift register while SS is high

19.6.2.6. Struct spi_slave_inst

SPI peripheral slave software instance structure, used to configure the correct SPI transfer mode settings for an attached slave. See [spi_select_slave](#).

Table 19-7. Members

Type	Name	Description
uint8_t	address	Address of slave device
bool	address_enabled	Address recognition enabled in slave device
uint8_t	ss_pin	Pin to use as slave select

19.6.2.7. Struct spi_slave_inst_config

SPI Peripheral slave configuration structure.

Table 19-8. Members

Type	Name	Description
uint8_t	address	Address of slave
bool	address_enabled	Enable address
uint8_t	ss_pin	Pin to use as slave select

19.6.3. Macro Definitions

19.6.3.1. Driver Feature Definition

Define SERCOM SPI features set according to different device family.

Macro FEATURE_SPI_SLAVE_SELECT_LOW_DETECT

```
#define FEATURE_SPI_SLAVE_SELECT_LOW_DETECT
```

SPI slave select low detection.

Macro FEATURE_SPI_HARDWARE_SLAVE_SELECT

```
#define FEATURE_SPI_HARDWARE_SLAVE_SELECT
```

Slave select can be controlled by hardware.

Macro FEATURE_SPI_ERROR_INTERRUPT

```
#define FEATURE_SPI_ERROR_INTERRUPT
```

SPI with error detect feature.

Macro FEATURE_SPI_SYNC_SCHEME_VERSION_2

```
#define FEATURE_SPI_SYNC_SCHEME_VERSION_2
```

SPI sync scheme version 2.

19.6.3.2. Macro PINMUX_DEFAULT

```
#define PINMUX_DEFAULT
```

Default pinmux.

19.6.3.3. Macro PINMUX_UNUSED

```
#define PINMUX_UNUSED
```

Unused pinmux.

19.6.3.4. Macro SPI_TIMEOUT

```
#define SPI_TIMEOUT
```

SPI timeout value.

19.6.4. Function Definitions

19.6.4.1. Driver Initialization and Configuration

Function spi_get_config_defaults()

Initializes an SPI configuration structure to default values.

```
void spi_get_config_defaults(  
    struct spi_config *const config)
```

This function will initialize a given SPI configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- Master mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- MUX Setting D
- Character size eight bits
- Not enabled in sleep mode
- Receiver enabled
- Baudrate 100000
- Default pinmux settings for all pads
- GCLK generator 0

Table 19-9. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `spi_slave_inst_get_config_defaults()`

Initializes an SPI peripheral slave device configuration structure to default values.

```
void spi_slave_inst_get_config_defaults(  
    struct spi_slave_inst_config *const config)
```

This function will initialize a given SPI slave device configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- Slave Select on GPIO pin 10
- Addressing not enabled

Table 19-10. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `spi_attach_slave()`

Attaches an SPI peripheral slave.

```
void spi_attach_slave(  
    struct spi_slave_inst *const slave,  
    const struct spi_slave_inst_config *const config)
```

This function will initialize the software SPI peripheral slave, based on the values of the config struct. The slave can then be selected and optionally addressed by the [spi_select_slave](#) function.

Table 19-11. Parameters

Data direction	Parameter name	Description
[out]	slave	Pointer to the software slave instance struct
[in]	config	Pointer to the config struct

Function `spi_init()`

Initializes the SERCOM SPI module.

```
enum status_code spi_init(  
    struct spi_module *const module,  
    Sercom *const hw,  
    const struct spi_config *const config)
```

This function will initialize the SERCOM SPI module, based on the values of the config struct.

Table 19-12. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Pointer to hardware instance
[in]	config	Pointer to the config struct

Returns

Status of the initialization.

Table 19-13. Return Values

Return value	Description
STATUS_OK	Module initiated correctly
STATUS_ERR_DENIED	If module is enabled
STATUS_BUSY	If module is busy resetting
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

19.6.4.2. Enable/Disable

Function `spi_enable()`

Enables the SERCOM SPI module.

```
void spi_enable(  
    struct spi_module *const module)
```

This function will enable the SERCOM SPI module.

Table 19-14. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `spi_disable()`

Disables the SERCOM SPI module.

```
void spi_disable(  
    struct spi_module *const module)
```

This function will disable the SERCOM SPI module.

Table 19-15. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

Function `spi_reset()`

Resets the SPI module.

```
void spi_reset(  
    struct spi_module *const module)
```

This function will reset the SPI module to its power on default values and disable it.

Table 19-16. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

19.6.4.3. Lock/Unlock

Function `spi_lock()`

Attempt to get lock on driver instance.

```
enum status_code spi_lock(  
    struct spi_module *const module)
```

This function checks the instance's lock, which indicates whether or not it is currently in use, and sets the lock if it was not already set.

The purpose of this is to enable exclusive access to driver instances, so that, e.g., transactions by different services will not interfere with each other.

Table 19-17. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 19-18. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

Function `spi_unlock()`

Unlock driver instance.

```
void spi_unlock(  
    struct spi_module *const module)
```

This function clears the instance lock, indicating that it is available for use.

Table 19-19. Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 19-20. Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

19.6.4.4. Ready to Write/Read

Function `spi_is_write_complete()`

Checks if the SPI in master mode has shifted out last data, or if the master has ended the transfer in slave mode.

```
bool spi_is_write_complete(  
    struct spi_module *const module)
```

This function will check if the SPI master module has shifted out last data, or if the slave select pin has been drawn high by the master for the SPI slave module.

Table 19-21. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Returns

Indication of whether any writes are ongoing.

Table 19-22. Return Values

Return value	Description
true	If the SPI master module has shifted out data, or slave select has been drawn high for SPI slave
false	If the SPI master module has not shifted out data

Function `spi_is_ready_to_write()`

Checks if the SPI module is ready to write data.

```
bool spi_is_ready_to_write(  
    struct spi_module *const module)
```

This function will check if the SPI module is ready to write data.

Table 19-23. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Returns

Indication of whether the module is ready to read data or not.

Table 19-24. Return Values

Return value	Description
true	If the SPI module is ready to write data
false	If the SPI module is not ready to write data

Function `spi_is_ready_to_read()`

Checks if the SPI module is ready to read data.

```
bool spi_is_ready_to_read(  
    struct spi_module *const module)
```

This function will check if the SPI module is ready to read data.

Table 19-25. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Returns

Indication of whether the module is ready to read data or not.

Table 19-26. Return Values

Return value	Description
true	If the SPI module is ready to read data
false	If the SPI module is not ready to read data

19.6.4.5. Read/Write**Function spi_write()**

Transfers a single SPI character.

```
enum status_code spi_write(
    struct spi_module * module,
    uint16_t tx_data)
```

This function will send a single SPI character via SPI and ignore any data shifted in by the connected device. To both send and receive data, use the [spi_transceive_wait](#) function or use the [spi_read](#) function after writing a character. The [spi_is_ready_to_write](#) function should be called before calling this function.

Note that this function does not handle the SS (Slave Select) pin(s) in master mode; this must be handled from the user application.

Note: In slave mode, the data will not be transferred before a master initiates a transaction.

Table 19-27. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Data to transmit

Returns

Status of the procedure.

Table 19-28. Return Values

Return value	Description
STATUS_OK	If the data was written
STATUS_BUSY	If the last write was not completed

Function `spi_write_buffer_wait()`

Sends a buffer of `length` SPI characters.

```
enum status_code spi_write_buffer_wait(  
    struct spi_module *const module,  
    const uint8_t * tx_data,  
    uint16_t length)
```

This function will send a buffer of SPI characters via the SPI and discard any data that is received. To both send and receive a buffer of data, use the [spi_transceive_buffer_wait](#) function.

Note that this function does not handle the `_SS` (slave select) pin(s) in master mode; this must be handled by the user application.

Table 19-29. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Pointer to the buffer to transmit
[in]	length	Number of SPI characters to transfer

Returns

Status of the write operation.

Table 19-30. Return Values

Return value	Description
STATUS_OK	If the write was completed
STATUS_ABORTED	If transaction was ended by master before entire buffer was transferred
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode

Function `spi_read()`

Reads last received SPI character.

```
enum status_code spi_read(  
    struct spi_module *const module,  
    uint16_t * rx_data)
```

This function will return the last SPI character shifted into the receive register by the [spi_write](#) function.

Note: The [spi_is_ready_to_read](#) function should be called before calling this function.

Note: Receiver must be enabled in the configuration.

Table 19-31. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[out]	rx_data	Pointer to store the received data

Returns

Status of the read operation.

Table 19-32. Return Values

Return value	Description
STATUS_OK	If data was read
STATUS_ERR_IO	If no data is available
STATUS_ERR_OVERFLOW	If the data is overflown

Function `spi_read_buffer_wait()`

Reads buffer of `length` SPI characters.

```
enum status_code spi_read_buffer_wait(  
    struct spi_module *const module,  
    uint8_t * rx_data,  
    uint16_t length,  
    uint16_t dummy)
```

This function will read a buffer of data from an SPI peripheral by sending dummy SPI character if in master mode, or by waiting for data in slave mode.

Note: If address matching is enabled for the slave, the first character received and placed in the buffer will be the address.

Table 19-33. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[out]	rx_data	Data buffer for received data
[in]	length	Length of data to receive
[in]	dummy	8- or 9-bit dummy byte to shift out in master mode

Returns

Status of the read operation.

Table 19-34. Return Values

Return value	Description
STATUS_OK	If the read was completed
STATUS_ABORTED	If transaction was ended by master before the entire buffer was transferred
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode

Return value	Description
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_OVERFLOW	If the data is overflown

Function `spi_transceive_wait()`

Sends and reads a single SPI character.

```
enum status_code spi_transceive_wait(
    struct spi_module *const module,
    uint16_t tx_data,
    uint16_t * rx_data)
```

This function will transfer a single SPI character via SPI and return the SPI character that is shifted into the shift register.

In master mode the SPI character will be sent immediately and the received SPI character will be read as soon as the shifting of the data is complete.

In slave mode this function will place the data to be sent into the transmit buffer. It will then block until an SPI master has shifted a complete SPI character, and the received data is available.

Note: The data to be sent might not be sent before the next transfer, as loading of the shift register is dependent on SCK.

Note: If address matching is enabled for the slave, the first character received and placed in the buffer will be the address.

Table 19-35. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	SPI character to transmit
[out]	rx_data	Pointer to store the received SPI character

Returns

Status of the operation.

Table 19-36. Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_OVERFLOW	If the incoming data is overflown

Function spi_transceive_buffer_wait()

Sends and receives a buffer of `length` SPI characters.

```
enum status_code spi_transceive_buffer_wait(  
    struct spi_module *const module,  
    uint8_t * tx_data,  
    uint8_t * rx_data,  
    uint16_t length)
```

This function will send and receive a buffer of data via the SPI.

In master mode the SPI characters will be sent immediately and the received SPI character will be read as soon as the shifting of the SPI character is complete.

In slave mode this function will place the data to be sent into the transmit buffer. It will then block until an SPI master has shifted the complete buffer and the received data is available.

Table 19-37. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Pointer to the buffer to transmit
[out]	rx_data	Pointer to the buffer where received data will be stored
[in]	length	Number of SPI characters to transfer

Returns

Status of the operation.

Table 19-38. Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_OVERFLOW	If the data is overflown

Function spi_select_slave()

Selects slave device.

```
enum status_code spi_select_slave(  
    struct spi_module *const module,  
    struct spi_slave_inst *const slave,  
    bool select)
```

This function will drive the slave select pin of the selected device low or high depending on the select Boolean. If slave address recognition is enabled, the address will be sent to the slave when selecting it.

Table 19-39. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software module struct
[in]	slave	Pointer to the attached slave
[in]	select	Boolean stating if the slave should be selected or deselected

Returns

Status of the operation.

Table 19-40. Return Values

Return value	Description
STATUS_OK	If the slave device was selected
STATUS_ERR_UNSUPPORTED_DEV	If the SPI module is operating in slave mode
STATUS_BUSY	If the SPI module is not ready to write the slave address

19.6.4.6. Callback Management**Function spi_register_callback()**

Registers a SPI callback function.

```
void spi_register_callback(
    struct spi_module *const module,
    spi_callback_t callback_func,
    enum spi_callback callback_type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled by [spi_enable_callback](#), in order for the interrupt handler to call it when the conditions for the callback type are met.

Table 19-41. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

Function spi_unregister_callback()

Unregisters a SPI callback function.

```
void spi_unregister_callback(
    struct spi_module * module,
    enum spi_callback callback_type)
```

Unregisters a callback function which is implemented by the user.

Table 19-42. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	callback_type	Callback type given by an enum

Function spi_enable_callback()

Enables an SPI callback of a given type.

```
void spi_enable_callback(
    struct spi_module *const module,
    enum spi_callback callback_type)
```

Enables the callback function registered by the [spi_register_callback](#). The callback function will be called from the interrupt handler when the conditions for the callback type are met.

Table 19-43. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	callback_type	Callback type given by an enum

Function spi_disable_callback()

Disables callback.

```
void spi_disable_callback(
    struct spi_module *const module,
    enum spi_callback callback_type)
```

Disables the callback function registered by the [spi_register_callback](#), and the callback will not be called from the interrupt routine.

Table 19-44. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	callback_type	Callback type given by an enum

19.6.4.7. Writing and Reading**Function spi_write_buffer_job()**

Asynchronous buffer write.

```
enum status_code spi_write_buffer_job(
    struct spi_module *const module,
    uint8_t * tx_data,
    uint16_t length)
```

Sets up the driver to write to the SPI from a given buffer. If registered and enabled, a callback function will be called when the write is finished.

Table 19-45. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[out]	tx_data	Pointer to data buffer to receive
[in]	length	Data buffer length

Returns

Status of the write request operation.

Table 19-46. Return Values

Return value	Description
STATUS_OK	If the operation completed successfully
STATUS_ERR_BUSY	If the SPI was already busy with a write operation
STATUS_ERR_INVALID_ARG	If requested write length was zero

Function spi_read_buffer_job()

Asynchronous buffer read.

```
enum status_code spi_read_buffer_job(
    struct spi_module *const module,
    uint8_t * rx_data,
    uint16_t length,
    uint16_t dummy)
```

Sets up the driver to read from the SPI to a given buffer. If registered and enabled, a callback function will be called when the read is finished.

Note: If address matching is enabled for the slave, the first character received and placed in the RX buffer will be the address.

Table 19-47. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[out]	rx_data	Pointer to data buffer to receive
[in]	length	Data buffer length
[in]	dummy	Dummy character to send when reading in master mode

Returns

Status of the operation.

Table 19-48. Return Values

Return value	Description
STATUS_OK	If the operation completed successfully
STATUS_ERR_BUSY	If the SPI was already busy with a read operation
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_INVALID_ARG	If requested read length was zero

Function spi_transceive_buffer_job()

Asynchronous buffer write and read.

```
enum status_code spi_transceive_buffer_job(
    struct spi_module *const module,
    uint8_t * tx_data,
    uint8_t * rx_data,
    uint16_t length)
```

Sets up the driver to write and read to and from given buffers. If registered and enabled, a callback function will be called when the transfer is finished.

Note: If address matching is enabled for the slave, the first character received and placed in the RX buffer will be the address.

Table 19-49. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	tx_data	Pointer to data buffer to send
[out]	rx_data	Pointer to data buffer to receive
[in]	length	Data buffer length

Returns

Status of the operation.

Table 19-50. Return Values

Return value	Description
STATUS_OK	If the operation completed successfully
STATUS_ERR_BUSY	If the SPI was already busy with a read operation
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_INVALID_ARG	If requested read length was zero

Function spi_abort_job()

Aborts an ongoing job.

```
void spi_abort_job(  
    struct spi_module *const module)
```

This function will abort the specified job type.

Table 19-51. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct

Function spi_get_job_status()

Retrieves the current status of a job.

```
enum status_code spi_get_job_status(  
    const struct spi_module *const module)
```

Retrieves the current status of a job that was previously issued.

Table 19-52. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct

Returns

Current job status.

Function spi_get_job_status_wait()

Retrieves the status of job once it ends.

```
enum status_code spi_get_job_status_wait(  
    const struct spi_module *const module)
```

Waits for current job status to become non-busy, then returns its value.

Table 19-53. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct

Returns

Current non-busy job status.

19.6.4.8. Function spi_is_syncing()

Determines if the SPI module is currently synchronizing to the bus.

```
bool spi_is_syncing(  
    struct spi_module *const module)
```

This function will check if the underlying hardware peripheral module is currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on the module until it is ready.

Table 19-54. Parameters

Data direction	Parameter name	Description
[in]	module	SPI hardware module

Returns

Synchronization status of the underlying hardware module.

Table 19-55. Return Values

Return value	Description
true	Module synchronization is ongoing
false	Module synchronization is not ongoing

19.6.4.9. Function `spi_set_baudrate()`

Set the baudrate of the SPI module.

```
enum status_code spi_set_baudrate(
    struct spi_module *const module,
    uint32_t baudrate)
```

This function will set the baudrate of the SPI module.

Table 19-56. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	baudrate	The baudrate wanted

Returns

The status of the configuration.

Table 19-57. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_OK	If the configuration was written

19.6.5. Enumeration Definitions

19.6.5.1. Enum `spi_addr_mode`

For slave mode when using the SPI frame with address format.

Table 19-58. Members

Enum value	Description
SPI_ADDR_MODE_MASK	address_mask in the spi_config struct is used as a mask to the register
SPI_ADDR_MODE_UNIQUE	The slave responds to the two unique addresses in address and address_mask in the spi_config struct
SPI_ADDR_MODE_RANGE	The slave responds to the range of addresses between and including address and address_mask in the spi_config struct

19.6.5.2. Enum spi_callback

Callbacks for SPI callback driver.

Note: For slave mode, these callbacks will be called when a transaction is ended by the master pulling Slave Select high.

Table 19-59. Members

Enum value	Description
SPI_CALLBACK_BUFFER_TRANSMITTED	Callback for buffer transmitted
SPI_CALLBACK_BUFFER_RECEIVED	Callback for buffer received
SPI_CALLBACK_BUFFER_TRANSCEIVED	Callback for buffers transceived
SPI_CALLBACK_ERROR	Callback for error
SPI_CALLBACK_SLAVE_TRANSMISSION_COMPLETE	Callback for transmission ended by master before the entire buffer was read or written from slave
SPI_CALLBACK_SLAVE_SELECT_LOW	Callback for slave select low
SPI_CALLBACK_COMBINED_ERROR	Callback for combined error happen

19.6.5.3. Enum spi_character_size

SPI character size.

Table 19-60. Members

Enum value	Description
SPI_CHARACTER_SIZE_8BIT	8-bit character
SPI_CHARACTER_SIZE_9BIT	9-bit character

19.6.5.4. Enum spi_data_order

SPI data order.

Table 19-61. Members

Enum value	Description
SPI_DATA_ORDER_LSB	The LSB of the data is transmitted first
SPI_DATA_ORDER_MSB	The MSB of the data is transmitted first

19.6.5.5. Enum spi_frame_format

Frame format for slave mode.

Table 19-62. Members

Enum value	Description
SPI_FRAME_FORMAT_SPI_FRAME	SPI frame
SPI_FRAME_FORMAT_SPI_FRAME_ADDR	SPI frame with address

19.6.5.6. Enum spi_interrupt_flag

Interrupt flags for the SPI module.

Table 19-63. Members

Enum value	Description
SPI_INTERRUPT_FLAG_DATA_REGISTER_EMPTY	This flag is set when the contents of the data register has been moved to the shift register and the data register is ready for new data
SPI_INTERRUPT_FLAG_TX_COMPLETE	This flag is set when the contents of the shift register has been shifted out
SPI_INTERRUPT_FLAG_RX_COMPLETE	This flag is set when data has been shifted into the data register
SPI_INTERRUPT_FLAG_SLAVE_SELECT_LOW	This flag is set when slave select low
SPI_INTERRUPT_FLAG_COMBINED_ERROR	This flag is set when combined error happen

19.6.5.7. Enum spi_mode

SPI mode selection.

Table 19-64. Members

Enum value	Description
SPI_MODE_MASTER	Master mode
SPI_MODE_SLAVE	Slave mode

19.6.5.8. Enum spi_signal_mux_setting

Set the functionality of the SERCOM pins. As not all combinations can be used in different modes of operation, proper combinations must be chosen according to the rest of the configuration.

Note: In master operation: DI is MISO, DO is MOSI. In slave operation: DI is MOSI, DO is MISO.

See [MUX Settings](#) for a description of the various MUX setting options.

Table 19-65. Members

Enum value	Description
SPI_SIGNAL_MUX_SETTING_A	SPI MUX combination A. DOPO: 0x0, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_B	SPI MUX combination B. DOPO: 0x0, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_C	SPI MUX combination C. DOPO: 0x0, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_D	SPI MUX combination D. DOPO: 0x0, DIPO: 0x3
SPI_SIGNAL_MUX_SETTING_E	SPI MUX combination E. DOPO: 0x1, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_F	SPI MUX combination F. DOPO: 0x1, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_G	SPI MUX combination G. DOPO: 0x1, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_H	SPI MUX combination H. DOPO: 0x1, DIPO: 0x3
SPI_SIGNAL_MUX_SETTING_I	SPI MUX combination I. DOPO: 0x2, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_J	SPI MUX combination J. DOPO: 0x2, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_K	SPI MUX combination K. DOPO: 0x2, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_L	SPI MUX combination L. DOPO: 0x2, DIPO: 0x3
SPI_SIGNAL_MUX_SETTING_M	SPI MUX combination M. DOPO: 0x3, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_N	SPI MUX combination N. DOPO: 0x3, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_O	SPI MUX combination O. DOPO: 0x3, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_P	SPI MUX combination P. DOPO: 0x3, DIPO: 0x3

19.6.5.9. Enum spi_transfer_mode

SPI transfer mode.

Table 19-66. Members

Enum value	Description
SPI_TRANSFER_MODE_0	Mode 0. Leading edge: rising, sample. Trailing edge: falling, setup
SPI_TRANSFER_MODE_1	Mode 1. Leading edge: rising, setup. Trailing edge: falling, sample
SPI_TRANSFER_MODE_2	Mode 2. Leading edge: falling, sample. Trailing edge: rising, setup
SPI_TRANSFER_MODE_3	Mode 3. Leading edge: falling, setup. Trailing edge: rising, sample

19.7. Extra Information for SERCOM SPI Driver

19.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
SERCOM	Serial Communication Interface
SPI	Serial Peripheral Interface
SCK	Serial Clock
MOSI	Master Output Slave Input
MISO	Master Input Slave Output
SS	Slave Select
DIO	Data Input Output
DO	Data Output
DI	Data Input
DMA	Direct Memory Access

19.7.2. Dependencies

The SPI driver has the following dependencies:

- System Pin Multiplexer Driver

19.7.3. Workarounds Implemented by Driver

No workarounds in driver.

19.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added new features as below: • Slave select low detect • Hardware slave select • DMA support
Edited slave part of write and transceive buffer functions to ensure that second character is sent at the right time

Changelog

Renamed the anonymous union in struct `spi_config` to `mode_specific`

Initial Release

19.8. Examples for SERCOM SPI Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Serial Peripheral Interface \(SERCOM SPI\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for SERCOM SPI Master - Polled](#)
- [Quick Start Guide for SERCOM SPI Slave - Polled](#)
- [Quick Start Guide for SERCOM SPI Master - Callback](#)
- [Quick Start Guide for SERCOM SPI Slave - Callback](#)
- [Quick Start Guide for Using DMA with SERCOM SPI](#)

19.8.1. Quick Start Guide for SERCOM SPI Master - Polled

In this use case, the SPI on extension header 1 of the Xplained Pro board will be configured with the following settings:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see [Master Mode Settings](#))
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

19.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

The following must be added to the user application:

A sample buffer to send via SPI.

```
static uint8_t buffer[BUF_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```


GPIO pin to use as Slave Select.

```
#define SLAVE_SELECT_PIN EXT1_PIN_SPI_SS_0
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_master_instance;
```

A globally available peripheral slave software device instance struct.

```
struct spi_slave_inst slave;
```

A function for configuring the SPI.

```
void configure_spi_master(void)
{
    struct spi_config config_spi_master;
    struct spi_slave_inst_config slave_dev_config;
    /* Configure and initialize software device instance of peripheral slave */
    spi_slave_inst_get_config_defaults(&slave_dev_config);
    slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
    spi_attach_slave(&slave, &slave_dev_config);
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_master);
    config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
    /* Configure pad 2 for data out */
    config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_master_instance, EXT1_SPI_MODULE, &config_spi_master);

    spi_enable(&spi_master_instance);
}
```

Add to user application main().

```
system_init();
configure_spi_master();
```

19.8.1.2. Workflow

1. Initialize system.

```
system_init();
```

2. Set-up the SPI.

```
configure_spi_master();
```

1. Create configuration struct.

```
struct spi_config config_spi_master;
```

2. Create peripheral slave configuration struct.

```
struct spi_slave_inst_config slave_dev_config;
```

3. Create peripheral slave software device instance struct.

```
struct spi_slave_inst slave;
```

4. Get default peripheral slave configuration.

```
spi_slave_inst_get_config_defaults(&slave_dev_config);
```

5. Set Slave Select pin.

```
slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
```

6. Initialize peripheral slave software instance with configuration.

```
spi_attach_slave(&slave, &slave_dev_config);
```

7. Get default configuration to edit.

```
spi_get_config_defaults(&config_spi_master);
```

8. Set MUX setting E.

```
config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

9. Set pinmux for pad 0 (data in (MISO)).

```
config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

10. Set pinmux for pad 1 as unused, so the pin can be used for other purposes.

```
config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
```

11. Set pinmux for pad 2 (data out (MOSI)).

```
config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

12. Set pinmux for pad 3 (SCK).

```
config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

13. Initialize SPI module with configuration.

```
spi_init(&spi_master_instance, EXT1_SPI_MODULE,  
&config_spi_master);
```

14. Enable SPI module.

```
spi_enable(&spi_master_instance);
```

19.8.1.3. Use Case

Code

Add the following to your user application `main()`.

```
while (true) {  
    /* Infinite loop */  
    if(!port_pin_get_input_level(BUTTON_0_PIN)) {  
        spi_select_slave(&spi_master_instance, &slave, true);  
        spi_write_buffer_wait(&spi_master_instance, buffer, BUF_LENGTH);  
        spi_select_slave(&spi_master_instance, &slave, false);  
        port_pin_set_output_level(LED_0_PIN, LED0_ACTIVE);  
    }  
}
```

Workflow

1. Select slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Write buffer to SPI slave.

```
spi_write_buffer_wait(&spi_master_instance, buffer, BUF_LENGTH);
```

3. Deselect slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

4. Light up.

```
port_pin_set_output_level(LED_0_PIN, LED0_ACTIVE);
```

5. Infinite loop.

```
while (true) {  
    /* Infinite loop */  
    if(!port_pin_get_input_level(BUTTON_0_PIN)) {  
        spi_select_slave(&spi_master_instance, &slave, true);  
        spi_write_buffer_wait(&spi_master_instance, buffer,  
BUF_LENGTH);  
        spi_select_slave(&spi_master_instance, &slave, false);  
        port_pin_set_output_level(LED_0_PIN, LED0_ACTIVE);  
    }  
}
```

19.8.2. Quick Start Guide for SERCOM SPI Slave - Polled

In this use case, the SPI on extension header 1 of the Xplained Pro board will be configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see [Slave Mode Settings](#))
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

19.8.2.1. Setup

Prerequisites

The device must be connected to an SPI master which must read from the device.

Code

The following must be added to the user application source file, outside any functions:

A sample buffer to send via SPI.

```
static uint8_t buffer_expect[BUF_LENGTH] = {  
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,  
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13  
};  
static uint8_t buffer_rx[BUF_LENGTH] = {0x00};
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_slave_instance;
```

A function for configuring the SPI.

```
void configure_spi_slave(void)
{
    struct spi_config config_spi_slave;
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_slave);
    config_spi_slave.mode = SPI_MODE_SLAVE;
    config_spi_slave.mode_specific.slave.preload_enable = true;
    config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
    config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
    /* Configure pad 2 for data out */
    config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);

    spi_enable(&spi_slave_instance);
}
```

Add to user application `main()`.

```
uint8_t result = 0;

/* Initialize system */
system_init();

configure_spi_slave();
```

Workflow

1. Initialize system.

```
system_init();
```

2. Set-up the SPI.

```
configure_spi_slave();
```

1. Create configuration struct.

```
struct spi_config config_spi_slave;
```

2. Get default configuration to edit.

```
spi_get_config_defaults(&config_spi_slave);
```

3. Set the SPI in slave mode.

```
config_spi_slave.mode = SPI_MODE_SLAVE;
```

4. Enable preloading of shift register.

```
config_spi_slave.mode_specific.slave.preload_enable = true;
```

5. Set frame format to SPI frame.

```
config_spi_slave.mode_specific.slave.frame_format =  
SPI_FRAME_FORMAT_SPI_FRAME;
```

6. Set MUX setting E.

```
config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

7. Set pinmux for pad 0 (data in MOSI).

```
config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

8. Set pinmux for pad 1 (slave select).

```
config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
```

9. Set pinmux for pad 2 (data out MISO).

```
config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

10. Set pinmux for pad 3 (SCK).

```
config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

11. Initialize SPI module with configuration.

```
spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);
```

12. Enable SPI module.

```
spi_enable(&spi_slave_instance);
```

19.8.2.2. Use Case

Code

Add the following to your user application main().

```
while(spi_read_buffer_wait(&spi_slave_instance, buffer_rx, BUF_LENGTH,  
0x00) != STATUS_OK) {  
    /* Wait for transfer from the master */  
}  
for (uint8_t i = 0; i < BUF_LENGTH; i++) {  
    if(buffer_rx[i] != buffer_expect[i]) {  
        result++;  
    }  
}  
while (true) {  
    /* Infinite loop */  
    if (result) {  
        port_pin_toggle_output_level(LED_0_PIN);  
        /* Add a short delay to see LED toggle */  
        volatile uint32_t delay = 30000;  
        while(delay--) {  
        }  
    } else {  
        port_pin_toggle_output_level(LED_0_PIN);  
        /* Add a short delay to see LED toggle */  
        volatile uint32_t delay = 600000;  
        while(delay--) {  
        }  
    }  
}
```

Workflow

1. Read data from SPI master.

```
while(spi_read_buffer_wait(&spi_slave_instance, buffer_rx, BUF_LENGTH,
    0x00) != STATUS_OK) {
    /* Wait for transfer from the master */
}
```

2. Compare the received data with the transmitted data from SPI master.

```
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}
```

3. Infinite loop. If the data is matched, LED0 will flash slowly. Otherwise, LED will flash quickly.

```
while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {}
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--) {}
    }
}
```

19.8.3. Quick Start Guide for SERCOM SPI Master - Callback

In this use case, the SPI on extension header 1 of the Xplained Pro board will be configured with the following settings:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see [Master Mode Settings](#))
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

19.8.3.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

The following must be added to the user application.

A sample buffer to send via SPI.

```
static uint8_t wr_buffer[BUF_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};
static uint8_t rd_buffer[BUF_LENGTH];
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

GPIO pin to use as Slave Select.

```
#define SLAVE_SELECT_PIN EXT1_PIN_SPI_SS_0
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_master_instance;
```

A globally available peripheral slave software device instance struct.

```
struct spi_slave_inst slave;
```

A function for configuring the SPI.

```
void configure_spi_master(void)
{
    struct spi_config config_spi_master;
    struct spi_slave_inst_config slave_dev_config;
    /* Configure and initialize software device instance of peripheral slave */
    spi_slave_inst_get_config_defaults(&slave_dev_config);
    slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
    spi_attach_slave(&slave, &slave_dev_config);
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_master);
    config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
    /* Configure pad 2 for data out */
    config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_master_instance, EXT1_SPI_MODULE, &config_spi_master);

    spi_enable(&spi_master_instance);
}
```

A function for configuring the callback functionality of the SPI.

```
void configure_spi_master_callbacks(void)
{
    spi_register_callback(&spi_master_instance, callback_spi_master,
        SPI_CALLBACK_BUFFER_TRANSCEIVED);
    spi_enable_callback(&spi_master_instance,
        SPI_CALLBACK_BUFFER_TRANSCEIVED);
}
```

A global variable that can flag to the application that the buffer has been transferred.

```
volatile bool transrev_complete_spi_master = false;
```

Callback function.

```
static void callback_spi_master( struct spi_module *const module)
{
    transrev_complete_spi_master = true;
}
```

Add to user application main().

```
/* Initialize system */
system_init();

configure_spi_master();
configure_spi_master_callbacks();
```

19.8.3.2. Workflow

1. Initialize system.

```
system_init();
```

2. Set-up the SPI.

```
configure_spi_master();
```

1. Create configuration struct.

```
struct spi_config config_spi_master;
```

2. Create peripheral slave configuration struct.

```
struct spi_slave_inst_config slave_dev_config;
```

3. Get default peripheral slave configuration.

```
spi_slave_inst_get_config_defaults(&slave_dev_config);
```

4. Set Slave Select pin.

```
slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
```

5. Initialize peripheral slave software instance with configuration.

```
spi_attach_slave(&slave, &slave_dev_config);
```

6. Get default configuration to edit.

```
spi_get_config_defaults(&config_spi_master);
```

7. Set MUX setting E.

```
config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

8. Set pinmux for pad 0 (data in MISO).

```
config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

9. Set pinmux for pad 1 as unused, so the pin can be used for other purposes.

```
config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
```

10. Set pinmux for pad 2 (data out MOSI).

```
config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

11. Set pinmux for pad 3 (SCK).

```
config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```


12. Initialize SPI module with configuration.

```
spi_init(&spi_master_instance, EXT1_SPI_MODULE,
&config_spi_master);
```

13. Enable SPI module.

```
spi_enable(&spi_master_instance);
```

3. Setup the callback functionality.

```
configure_spi_master_callbacks();
```

1. Register callback function for buffer transmitted.

```
spi_register_callback(&spi_master_instance, callback_spi_master,
SPI_CALLBACK_BUFFER_TRANSCIEVED);
```

2. Enable callback for buffer transmitted.

```
spi_enable_callback(&spi_master_instance,
SPI_CALLBACK_BUFFER_TRANSCIEVED);
```

19.8.3.3. Use Case

Code

Add the following to your user application `main()`.

```
while (true) {
    /* Infinite loop */
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {
        spi_select_slave(&spi_master_instance, &slave, true);
        spi_transceive_buffer_job(&spi_master_instance,
wr_buffer, rd_buffer, BUF_LENGTH);
        while (!transrev_complete_spi_master) {
        }
        transrev_complete_spi_master = false;
        spi_select_slave(&spi_master_instance, &slave, false);
    }
}
```

Workflow

1. Select slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Write buffer to SPI slave.

```
spi_transceive_buffer_job(&spi_master_instance,
wr_buffer, rd_buffer, BUF_LENGTH);
```

3. Wait for the transfer to be complete.

```
while (!transrev_complete_spi_master) {
}
transrev_complete_spi_master = false;
```

4. Deselect slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

5. Infinite loop.

```
while (true) {
    /* Infinite loop */
```

```

        if (!port_pin_get_input_level(BUTTON_0_PIN)) {
            spi_select_slave(&spi_master_instance, &slave, true);
            spi_transceive_buffer_job(&spi_master_instance,
                                     wr_buffer, rd_buffer, BUF_LENGTH);
            while (!transrev_complete_spi_master) {
            }
            transrev_complete_spi_master = false;
            spi_select_slave(&spi_master_instance, &slave, false);
        }
    }
}

```

19.8.3.4. Callback

When the buffer is successfully transmitted to the slave, the callback function will be called.

Workflow

1. Let the application know that the buffer is transmitted by setting the global variable to true.

```
transrev_complete_spi_master = true;
```

19.8.4. Quick Start Guide for SERCOM SPI Slave - Callback

In this use case, the SPI on extension header 1 of the Xplained Pro board will be configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see [Slave Mode Settings](#))
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

19.8.4.1. Setup

Prerequisites

The device must be connected to a SPI master, which must read from the device.

Code

The following must be added to the user application source file, outside any functions.

A sample buffer to send via SPI.

```

static uint8_t buffer_rx[BUF_LENGTH] = {0x00,};
static uint8_t buffer_expect[BUF_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};

```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_slave_instance;
```

A function for configuring the SPI.

```
void configure_spi_slave(void)
{
    struct spi_config config_spi_slave;
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_slave);
    config_spi_slave.mode = SPI_MODE_SLAVE;
    config_spi_slave.mode_specific.slave.preload_enable = true;
    config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
    config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
    /* Configure pad 2 for data out */
    config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);

    spi_enable(&spi_slave_instance);
}
```

A function for configuring the callback functionality of the SPI.

```
void configure_spi_slave_callbacks(void)
{
    spi_register_callback(&spi_slave_instance, spi_slave_callback,
        SPI_CALLBACK_BUFFER_RECEIVED);
    spi_enable_callback(&spi_slave_instance, SPI_CALLBACK_BUFFER_RECEIVED);
}
```

A global variable that can flag to the application that the buffer has been transferred.

```
volatile bool transfer_complete_spi_slave = false;
```

Callback function.

```
static void spi_slave_callback(struct spi_module *const module)
{
    transfer_complete_spi_slave = true;
}
```

Add to user application main().

```
uint8_t result = 0;

/* Initialize system */
system_init();

configure_spi_slave();
configure_spi_slave_callbacks();
```

Workflow

1. Initialize system.

```
system_init();
```

2. Set-up the SPI.

```
configure_spi_slave();
```

1. Create configuration struct.

```
struct spi_config config_spi_slave;
```

2. Get default configuration to edit.

```
spi_get_config_defaults(&config_spi_slave);
```

3. Set the SPI in slave mode.

```
config_spi_slave.mode = SPI_MODE_SLAVE;
```

4. Enable preloading of shift register.

```
config_spi_slave.mode_specific.slave.preload_enable = true;
```

5. Set frame format to SPI frame.

```
config_spi_slave.mode_specific.slave.frame_format =  
SPI_FRAME_FORMAT_SPI_FRAME;
```

6. Set MUX setting E.

```
config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

7. Set pinmux for pad 0 (data in MOSI).

```
config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

8. Set pinmux for pad 1 (slave select).

```
config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
```

9. Set pinmux for pad 2 (data out MISO).

```
config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

10. Set pinmux for pad 3 (SCK).

```
config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

11. Initialize SPI module with configuration.

```
spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);
```

12. Enable SPI module.

```
spi_enable(&spi_slave_instance);
```

3. Setup of the callback functionality.

```
configure_spi_slave_callbacks();
```

1. Register callback function for buffer transmitted.

```
spi_register_callback(&spi_slave_instance, spi_slave_callback,  
SPI_CALLBACK_BUFFER_RECEIVED);
```

2. Enable callback for buffer transmitted.

```
spi_enable_callback(&spi_slave_instance,  
SPI_CALLBACK_BUFFER_RECEIVED);
```

19.8.4.2. Use Case

Code

Add the following to your user application `main()`.

```
spi_read_buffer_job(&spi_slave_instance, buffer_rx, BUF_LENGTH, 0x00);
while(!transfer_complete_spi_slave) {
    /* Wait for transfer from master */
}
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}

while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {
        }
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--) {
        }
    }
}
```

Workflow

1. Initiate a read buffer job.

```
spi_read_buffer_job(&spi_slave_instance, buffer_rx, BUF_LENGTH, 0x00);
```

2. Wait for the transfer to be complete.

```
while(!transfer_complete_spi_slave) {
    /* Wait for transfer from master */
}
```

3. Compare the received data with the transmitted data from SPI master.

```
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}
```

4. Infinite loop. If the data is matched, LED0 will flash slowly. Otherwise, LED will flash quickly.

```
while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {
        }
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
    }
}
```

```

        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--){
        }
    }
}

```

19.8.4.3. Callback

When the buffer is successfully transmitted from the master, the callback function will be called.

Workflow

1. Let the application know that the buffer is transmitted by setting the global variable to true.

```
transfer_complete_spi_slave = true;
```

19.8.5. Quick Start Guide for Using DMA with SERCOM SPI

The supported board list:

- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM DA1 Xplained Pro
- SAM C21 Xplained Pro

This quick start will transmit a buffer data from master to slave through DMA. In this use case the SPI master will be configured with the following settings on SAM Xplained Pro:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

The SPI slave will be configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

Note that the pinouts on other boards may differ, see next sector for details.

19.8.5.1. Setup

Prerequisites

The following connections has to be made using wires:

- SAM D21/DA1 Xplained Pro.
 - **SS_0**: EXT1 PIN15 (PA05) <> EXT2 PIN15 (PA17)
 - **DO/DI**: EXT1 PIN16 (PA06) <> EXT2 PIN17 (PA16)
 - **DI/DO**: EXT1 PIN17 (PA04) <> EXT2 PIN16 (PA18)
 - **SCK**: EXT1 PIN18 (PA07) <> EXT2 PIN18 (PA19)
- SAM R21 Xplained Pro.
 - **SS_0**: EXT1 PIN15 (PB03) <> EXT1 PIN10 (PA23)
 - **DO/DI**: EXT1 PIN16 (PB22) <> EXT1 PIN9 (PA22)
 - **DI/DO**: EXT1 PIN17 (PB02) <> EXT1 PIN7 (PA18)
 - **SCK**: EXT1 PIN18 (PB23) <> EXT1 PIN8 (PA19)
- SAM L21 Xplained Pro.
 - **SS_0**: EXT1 PIN15 (PA05) <> EXT1 PIN12 (PA09)
 - **DO/DI**: EXT1 PIN16 (PA06) <> EXT1 PIN11 (PA08)
 - **DI/DO**: EXT1 PIN17 (PA04) <> EXT2 PIN03 (PA10)
 - **SCK**: EXT1 PIN18 (PA07) <> EXT2 PIN04 (PA11)
- SAM L22 Xplained Pro.
 - **SS_0**: EXT1 PIN15 (PB21) <> EXT2 PIN15 (PA17)
 - **DO/DI**: EXT1 PIN16 (PB00) <> EXT2 PIN17 (PA16)
 - **DI/DO**: EXT1 PIN17 (PB02) <> EXT2 PIN16 (PA18)
 - **SCK**: EXT1 PIN18 (PB01) <> EXT2 PIN18 (PA19)
- SAM C21 Xplained Pro.
 - **SS_0**: EXT1 PIN15 (PA17) <> EXT2 PIN15 (PB03)
 - **DO/DI**: EXT1 PIN16 (PA18) <> EXT2 PIN17 (PB02)
 - **DI/DO**: EXT1 PIN17 (PA16) <> EXT2 PIN16 (PB00)
 - **SCK**: EXT1 PIN18 (PA19) <> EXT2 PIN18 (PB01)

Code

Add to the main application source file, before user definitions and functions according to your board:

For SAM D21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE    EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN        EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING   EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0    EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1    PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2    EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3    EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE     EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING    EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0    EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1    EXT1_SPI_SERCOM_PINMUX_PAD1
```

```
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX SERCOM1_DMACH_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX SERCOM0_DMACH_ID_RX
```

For SAM R21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE SERCOM3
#define CONF_MASTER_SS_PIN EXT1_PIN_10
#define CONF_MASTER_MUX_SETTING SPI_SIGNAL_MUX_SETTING_E
#define CONF_MASTER_PINMUX_PAD0 PINMUX_PA22C_SERCOM3_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 PINMUX_PA18D_SERCOM3_PAD2
#define CONF_MASTER_PINMUX_PAD3 PINMUX_PA19D_SERCOM3_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX SERCOM3_DMACH_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX SERCOM5_DMACH_ID_RX
```

For SAM L21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE SERCOM2
#define CONF_MASTER_SS_PIN EXT1_PIN_12
#define CONF_MASTER_MUX_SETTING SPI_SIGNAL_MUX_SETTING_E
#define CONF_MASTER_PINMUX_PAD0 PINMUX_PA08D_SERCOM2_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 PINMUX_PA10D_SERCOM2_PAD2
#define CONF_MASTER_PINMUX_PAD3 PINMUX_PA11D_SERCOM2_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX SERCOM2_DMACH_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX SERCOM0_DMACH_ID_RX
```

For SAM L22 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
```



```
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX EXT2_SPI_SERCOM_DMACH_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX EXT1_SPI_SERCOM_DMACH_ID_RX
```

For SAM DA1 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX SERCOM1_DMACH_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX SERCOM0_DMACH_ID_RX
```

For SAM C21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX SERCOM5_DMACH_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX SERCOM1_DMACH_ID_RX
```

Add to the main application source file, outside of any functions:

```
#define BUF_LENGTH 20
```

```
#define TEST_SPI_BAUDRATE 1000000UL
```

```
#define SLAVE_SELECT_PIN CONF_MASTER_SS_PIN
```

```
static const uint8_t buffer_tx[BUF_LENGTH] = {
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
    0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14,
```

```
};
static uint8_t buffer_rx[BUF_LENGTH];

struct spi_module spi_master_instance;
struct spi_module spi_slave_instance;

static volatile bool transfer_tx_is_done = false;
static volatile bool transfer_rx_is_done = false;

struct spi_slave_inst slave;

COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_tx SECTION_DMAC_DESCRIPTOR;
DmacDescriptor example_descriptor_rx SECTION_DMAC_DESCRIPTOR;
```

Copy-paste the following setup code to your user application:

```
static void transfer_tx_done(struct dma_resource* const resource )
{
    transfer_tx_is_done = true;
}

static void transfer_rx_done(struct dma_resource* const resource )
{
    transfer_rx_is_done = true;
}

static void configure_dma_resource_tx(struct dma_resource *tx_resource)
{
    struct dma_resource_config tx_config;

    dma_get_config_defaults(&tx_config);

    tx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_TX;
    tx_config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(tx_resource, &tx_config);
}

static void configure_dma_resource_rx(struct dma_resource *rx_resource)
{
    struct dma_resource_config rx_config;

    dma_get_config_defaults(&rx_config);

    rx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_RX;
    rx_config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(rx_resource, &rx_config);
}

static void setup_transfer_descriptor_tx(DmacDescriptor *tx_descriptor)
{
    struct dma_descriptor_config tx_descriptor_config;

    dma_descriptor_get_config_defaults(&tx_descriptor_config);

    tx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    tx_descriptor_config.dst_increment_enable = false;
    tx_descriptor_config.block_transfer_count = sizeof(buffer_tx) /
sizeof(uint8_t);
    tx_descriptor_config.source_address = (uint32_t)buffer_tx +
sizeof(buffer_tx);
```

```

        tx_descriptor_config.destination_address =
            (uint32_t)(&spi_master_instance.hw->SPI.DATA.reg);

        dma_descriptor_create(tx_descriptor, &tx_descriptor_config);
    }

static void setup_transfer_descriptor_rx(DmacDescriptor *rx_descriptor)
{
    struct dma_descriptor_config rx_descriptor_config;

    dma_descriptor_get_config_defaults(&rx_descriptor_config);

    rx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    rx_descriptor_config.src_increment_enable = false;
    rx_descriptor_config.block_transfer_count = sizeof(buffer_rx) /
sizeof(uint8_t);
    rx_descriptor_config.source_address =
        (uint32_t)(&spi_slave_instance.hw->SPI.DATA.reg);
    rx_descriptor_config.destination_address =
        (uint32_t)buffer_rx + sizeof(buffer_rx);

    dma_descriptor_create(rx_descriptor, &rx_descriptor_config);
}

static void configure_spi_master(void)
{
    struct spi_config config_spi_master;
    struct spi_slave_inst_config slave_dev_config;
    /* Configure and initialize software device instance of peripheral slave */
    spi_slave_inst_get_config_defaults(&slave_dev_config);
    slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
    spi_attach_slave(&slave, &slave_dev_config);
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_master);
    config_spi_master.mode_specific.master.baudrate = TEST_SPI_BAUDRATE;
    config_spi_master.mux_setting = CONF_MASTER_MUX_SETTING;

    config_spi_master.pinmux_pad0 = CONF_MASTER_PINMUX_PAD0;
    config_spi_master.pinmux_pad1 = CONF_MASTER_PINMUX_PAD1;
    config_spi_master.pinmux_pad2 = CONF_MASTER_PINMUX_PAD2;
    config_spi_master.pinmux_pad3 = CONF_MASTER_PINMUX_PAD3;

    spi_init(&spi_master_instance, CONF_MASTER_SPI_MODULE,
&config_spi_master);

    spi_enable(&spi_master_instance);
}

static void configure_spi_slave(void)
{
    struct spi_config config_spi_slave;

    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_slave);
    config_spi_slave.mode = SPI_MODE_SLAVE;
    config_spi_slave.mode_specific.slave.preload_enable = true;
    config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
    config_spi_slave.mux_setting = CONF_SLAVE_MUX_SETTING;

    config_spi_slave.pinmux_pad0 = CONF_SLAVE_PINMUX_PAD0;
    config_spi_slave.pinmux_pad1 = CONF_SLAVE_PINMUX_PAD1;
    config_spi_slave.pinmux_pad2 = CONF_SLAVE_PINMUX_PAD2;

```

```

    config_spi_slave.pinmux_pad3 = CONF_SLAVE_PINMUX_PAD3;

    spi_init(&spi_slave_instance, CONF_SLAVE_SPI_MODULE,
&config_spi_slave);

    spi_enable(&spi_slave_instance);

}

```

Add to user application initialization (typically the start of `main()`):

```

configure_spi_master();
configure_spi_slave();

configure_dma_resource_tx(&example_resource_tx);
configure_dma_resource_rx(&example_resource_rx);

setup_transfer_descriptor_tx(&example_descriptor_tx);
setup_transfer_descriptor_rx(&example_descriptor_rx);

dma_add_descriptor(&example_resource_tx, &example_descriptor_tx);
dma_add_descriptor(&example_resource_rx, &example_descriptor_rx);

dma_register_callback(&example_resource_tx, transfer_tx_done,
DMA_CALLBACK_TRANSFER_DONE);
dma_register_callback(&example_resource_rx, transfer_rx_done,
DMA_CALLBACK_TRANSFER_DONE);

dma_enable_callback(&example_resource_tx, DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_rx, DMA_CALLBACK_TRANSFER_DONE);

```

Workflow

1. Create a module software instance structure for the SPI module to store the SPI driver state while it is in use.

```

struct spi_module spi_master_instance;
struct spi_module spi_slave_instance;

```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a module software instance structure for DMA resource to store the DMA resource state while it is in use.

```

struct dma_resource example_resource_tx;
struct dma_resource example_resource_rx;

```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

3. Create transfer done flag to indication DMA transfer done.

```

static volatile bool transfer_tx_is_done = false;
static volatile bool transfer_rx_is_done = false;

```

4. Define the buffer length for TX/RX.

```

#define BUF_LENGTH 20

```

5. Create buffer to store the data to be transferred.

```

static const uint8_t buffer_tx[BUF_LENGTH] = {
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
    0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14,

```

```
};
static uint8_t buffer_rx[BUF_LENGTH];
```

6. Create the SPI module configuration struct, which can be filled out to adjust the configuration of a physical SPI peripheral.

```
struct spi_config config_spi_master;
```

```
struct spi_config config_spi_slave;
```

7. Initialize the SPI configuration struct with the module's default values.

```
spi_get_config_defaults(&config_spi_master);
```

```
spi_get_config_defaults(&config_spi_slave);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

8. Alter the SPI settings to configure the physical pinout, baudrate, and other relevant parameters.

```
config_spi_master.mux_setting = CONF_MASTER_MUX_SETTING;
```

```
config_spi_slave.mux_setting = CONF_SLAVE_MUX_SETTING;
```

9. Configure the SPI module with the desired settings, retrying while the driver is busy until the configuration is stressfully set.

```
spi_init(&spi_master_instance, CONF_MASTER_SPI_MODULE,
&config_spi_master);
```

```
spi_init(&spi_slave_instance, CONF_SLAVE_SPI_MODULE,
&config_spi_slave);
```

10. Enable the SPI module.

```
spi_enable(&spi_master_instance);
```

```
spi_enable(&spi_slave_instance);
```

11. Create the DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config tx_config;
```

```
struct dma_resource_config rx_config;
```

12. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&tx_config);
```

```
dma_get_config_defaults(&rx_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

13. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM TX empty and RX complete trigger causes a beat transfer in this example.

```
tx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_TX;
tx_config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

```
rx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_RX;
rx_config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

14. Allocate a DMA resource with the configurations.

```
dma_allocate(tx_resource, &tx_config);
```

```
dma_allocate(rx_resource, &rx_config);
```

15. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config tx_descriptor_config;
```

```
struct dma_descriptor_config rx_descriptor_config;
```

16. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&tx_descriptor_config);
```

```
dma_descriptor_get_config_defaults(&rx_descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

17. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
tx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
tx_descriptor_config.dst_increment_enable = false;
tx_descriptor_config.block_transfer_count = sizeof(buffer_tx) /
sizeof(uint8_t);
tx_descriptor_config.source_address = (uint32_t)buffer_tx +
sizeof(buffer_tx);
tx_descriptor_config.destination_address =
(uint32_t)(&spi_master_instance.hw->SPI.DATA.reg);
```

```
rx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
rx_descriptor_config.src_increment_enable = false;
rx_descriptor_config.block_transfer_count = sizeof(buffer_rx) /
sizeof(uint8_t);
rx_descriptor_config.source_address =
(uint32_t)(&spi_slave_instance.hw->SPI.DATA.reg);
rx_descriptor_config.destination_address =
(uint32_t)buffer_rx + sizeof(buffer_rx);
```

18. Create the DMA transfer descriptor.

```
dma_descriptor_create(tx_descriptor, &tx_descriptor_config);
```

```
dma_descriptor_create(rx_descriptor, &rx_descriptor_config);
```

19.8.5.2. Use Case

Code

Copy-paste the following code to your user application:

```
spi_select_slave(&spi_master_instance, &slave, true);

dma_start_transfer_job(&example_resource_rx);
dma_start_transfer_job(&example_resource_tx);

while (!transfer_rx_is_done) {
    /* Wait for transfer done */
}

spi_select_slave(&spi_master_instance, &slave, false);

while (true) {
}
```

Workflow

1. Select the slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Start the transfer job.

```
dma_start_transfer_job(&example_resource_rx);
dma_start_transfer_job(&example_resource_tx);
```

3. Wait for transfer done.

```
while (!transfer_rx_is_done) {
    /* Wait for transfer done */
}
```

4. Deselect the slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

5. Enter endless loop.

```
while (true) {
}
```

19.9. MUX Settings

The following lists the possible internal SERCOM module pad function assignments for the four SERCOM pads in both SPI Master and SPI Slave modes. They are combinations of DOPO and DIPO in CTRLA.

Note that this is in addition to the physical GPIO pin MUX of the device, and can be used in conjunction to optimize the serial data pin-out.

19.9.1. Master Mode Settings

The following table describes the SERCOM pin functionalities for the various MUX settings, whilst in SPI Master mode.

Note: If MISO is unlisted, the SPI receiver must not be enabled for the given MUX setting.

Combination	DOPO / DIPO	SERCOM PAD[0]	SERCOM PAD[1]	SERCOM PAD[2]	SERCOM PAD[3]
A	0x0 / 0x0	MOSI	SCK	-	-
B	0x0 / 0x1	MOSI	SCK	-	-
C	0x0 / 0x2	MOSI	SCK	MISO	-
D	0x0 / 0x3	MOSI	SCK	-	MISO
E	0x1 / 0x0	MISO	-	MOSI	SCK
F	0x1 / 0x1	-	MISO	MOSI	SCK
G	0x1 / 0x2	-	-	MOSI	SCK
H	0x1 / 0x3	-	-	MOSI	SCK
I	0x2 / 0x0	MISO	SCK	-	MOSI
J	0x2 / 0x1	-	SCK	-	MOSI
K	0x2 / 0x2	-	SCK	MISO	MOSI
L	0x2 / 0x3	-	SCK	-	MOSI
M	0x3 / 0x0	MOSI	-	-	SCK
N	0x3 / 0x1	MOSI	MISO	-	SCK
O	0x3 / 0x2	MOSI	-	MISO	SCK
P	0x3 / 0x3	MOSI	-	-	SCK

19.9.2. Slave Mode Settings

The following table describes the SERCOM pin functionalities for the various MUX settings, whilst in SPI Slave mode.

Note: If MISO is unlisted, the SPI receiver must not be enabled for the given MUX setting.

Combination	DOPO / DIPO	SERCOM PAD[0]	SERCOM PAD[1]	SERCOM PAD[2]	SERCOM PAD[3]
A	0x0 / 0x0	MISO	SCK	/SS	-
B	0x0 / 0x1	MISO	SCK	/SS	-
C	0x0 / 0x2	MISO	SCK	/SS	-
D	0x0 / 0x3	MISO	SCK	/SS	MOSI
E	0x1 / 0x0	MOSI	/SS	MISO	SCK
F	0x1 / 0x1	-	/SS	MISO	SCK

Combination	DOPO / DIPO	SERCOM PAD[0]	SERCOM PAD[1]	SERCOM PAD[2]	SERCOM PAD[3]
G	0x1 / 0x2	-	/SS	MISO	SCK
H	0x1 / 0x3	-	/SS	MISO	SCK
I	0x2 / 0x0	MOSI	SCK	/SS	MISO
J	0x2 / 0x1	-	SCK	/SS	MISO
K	0x2 / 0x2	-	SCK	/SS	MISO
L	0x2 / 0x3	-	SCK	/SS	MISO
M	0x3 / 0x0	MISO	/SS	-	SCK
N	0x3 / 0x1	MISO	/SS	-	SCK
O	0x3 / 0x2	MISO	/SS	MOSI	SCK
P	0x3 / 0x3	MISO	/SS	-	SCK

20. SAM Segment Liquid Crystal Display(SLCD) Controller

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's SLCD functionality. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- Segment Liquid Crystal Display(SLCD)

The following devices can use this module:

- Atmel | SMART SAM L22

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

20.1. Prerequisites

There are no prerequisites for this module.

20.2. Module Overview

20.2.1. Display Overview

A LCD display is made of several segments (pixels or complete symbols) which can be visible or invisible. A segment has two electrodes with liquid crystal between them. These electrodes are the common terminal (COM pin) and the segment terminal (SEG pin). When a voltage above a threshold voltage is applied across the liquid crystal, the segment becomes visible. The voltage must alternate, to avoid an electrophoresis effect in the liquid crystal, which degrades the display.

The LCD controller is intended for monochrome passive liquid crystal display (LCD) with up to 8 common terminals and up to 44 segment terminals. A charge pump provides LCD display supply which can be higher than supply voltage of the device. Each LCD pin, segment or common terminals, can be configured as general purpose I/O pins if not driven by LCD controller.

20.3. Special Considerations

20.3.1. I/O Lines

The SLCD pins (SEG and COM) are multiplexed with other peripherals. The user application must first configure the I/O controller, to give control of the requisite pins to the SLCD.

20.3.2. Power Management

The SLCD will continue to operate in any sleep mode where the selected source clock is running. The SLCD interrupts can be used to wake up the device from sleep modes. Events connected to the event system can trigger other operations in the system without exiting sleep modes.

The power consumption of SLCD itself can be minimized by:

- Using the lowest acceptable frame rate (refer to the LCD glass technical characteristics)
- Using the low-power waveform (default mode)
- Using automated modes of operation
- Configuring the lowest possible contrast value

20.4. Extra Information

For extra information, see [Extra Information for SLCD](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

20.5. Examples

For a list of examples related to this driver, see [Examples for SLCD](#).

20.6. API Overview

20.6.1. Variable and Type Definitions

20.6.1.1. Type `slcd_callback_t`

```
typedef void(* slcd_callback_t )(enum slcd_callback_type type)
```

SLCD interrupt callback function type.

20.6.2. Structure Definitions

20.6.2.1. Struct `slcd_automated_char_config`

SLCD automated char configuration.

Table 20-1. Members

Type	Name	Description
uint8_t	com_line_num	Define the number of COM line per row, it equal to number of COM line - 1
uint32_t	data_mask	Segments data mask
uint8_t	digit_num	Define the number of digit, it must be greater than 1

Type	Name	Description
enum slcd_frame_counter	fc	Frame counter selection for automated character mapping
enum slcd_automated_char_mode	mode	Display mode
enum slcd_automated_char_order	order	Mapping order in automated char mode
uint8_t	row_digit_num	Define the number of digit per row
uint8_t	scrolling_step	Define the number of steps in scrolling mode. scrolling_step = character string length - digit_num + 1
uint8_t	seg_line_num	Define the number of SEG line per digit, it equal to number of SEG line - 1
uint8_t	start_seg_line	Define the index of the first segment terminal of the digit to display

20.6.2.2. Struct `slcd_blink_config`

SLCD blink configuration.

Table 20-2. Members

Type	Name	Description
bool	blink_all_seg	All segments are allowed to blink if true, else only Selected segments are allowed to blink
enum slcd_frame_counter	fc	Frame counter selection for blinking

20.6.2.3. Struct `slcd_circular_shift_config`

SLCD circular shift configuration.

Table 20-3. Members

Type	Name	Description
uint16_t	data	Circular shift register value
enum slcd_circular_shift_dir	dir	Shift direction
enum slcd_frame_counter	fc	Frame counter selection for circular shift
uint8_t	size	Size of the circular shift register, MAX. size is 16

20.6.2.4. Struct `slcd_config`

Basic configuration for SLCDC.

Table 20-4. Members

Type	Name	Description
uint8_t	bias_buffer_duration	Bias buffer duration
bool	enable_bias_buffer	Enable bias buffer if true
bool	enable_ext_bias	Enable external bias capacitor if true
bool	enable_low_resistance	Enable Low resistance if true
uint8_t	low_resistance_duration	Low resistance network duration
bool	run_in_standby	Keep SLCD enabled in standby sleep mode if true
enum slcd_waveform_mode	waveform_mode	waveform mode selection

20.6.2.5. Struct slcd_events

Event flags for the SLCD module. This is used to enable and disable events via [slcd_enable_events\(\)](#) and [slcd_disable_events\(\)](#).

Table 20-5. Members

Type	Name	Description
bool	generate_event_on_fc0_overflow	Enable event generation on frame counter 0 overflow
bool	generate_event_on_fc1_overflow	Enable event generation on frame counter 1 overflow
bool	generate_event_on_fc2_overflow	Enable event generation on frame counter 2 overflow

20.6.3. Macro Definitions

20.6.3.1. Macro SLCD_CALLBACK_TYPE_NUM

```
#define SLCD_CALLBACK_TYPE_NUM
```

20.6.4. Function Definitions

20.6.4.1. SLCD Basic Operation Functions

Function slcd_get_config_defaults()

Initializes SLCD configurations struct to defaults.

```
void slcd_get_config_defaults(  
    struct slcd_config * config)
```

Initializes SLCD configuration struct to predefined safe default settings.

Table 20-6. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an instance of struct slcd_config

Function slcd_init()

Initialize SLCD module.

```
enum status_code slcd_init(  
    struct slcd_config *const config)
```

Table 20-7. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an SLCD configuration structure

Note: SLCD cannot be Initialized while it is enabled.

Returns

Status of the configuration procedure.

Table 20-8. Return Values

Return value	Description
STATUS_OK	SLCD configuration went successful
STATUS_ERR_INVALID_ARG	If an invalid configuration was supplied

Function slcd_enable()

Enables the SLCD module.

```
void slcd_enable( void )
```

Enables the SLCD module once it has been configured, ready for use. Most module configuration parameters cannot be altered while the module is enabled.

Function slcd_disable()

Disables the SLCD module.

```
void slcd_disable( void )
```

Disables the SLCD module.

Function slcd_is_enabled()

Check if SLCD module is enabled or not.

```
bool slcd_is_enabled( void )
```

Check if SLCD module is enabled or not.

Returns

Enable status.

Table 20-9. Return Values

Return value	Description
true	SLCD module is enabled
false	SLCD module is disabled

Function slcd_reset()

Reset the SLCD module.

```
void slcd_reset( void )
```

Reset the SLCD module.

Function slcd_set_contrast()

Set the SLCD fine contrast.

```
enum status_code slcd_set_contrast(
    uint8_t contrast)
```

The LCD contrast is defined by the value of VLCD voltage. The higher is the VLCD voltage, the higher is the contrast. The software contrast adjustment is only possible in internal supply mode. In internal supply mode, VLCD is in the range 2.5V to 3.5V. VLCD can be adjusted with 16 steps, each step is 60 mV. The contrast value can be written at any time.

Table 20-10. Parameters

Data direction	Parameter name	Description
[in]	contrast	Contrast value

Returns

Status of set contrast.

Table 20-11. Return Values

Return value	Description
STATUS_OK	SLCD contrast set successful
STATUS_ERR_INVALID_ARG	SLCD is not working in internal supply mode

Function slcd_is_syncing()

Determines if SLCD module is currently synchronizing to the bus.

```
bool slcd_is_syncing( void )
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Returns

Synchronization status of the underlying hardware module.

Table 20-12. Return Values

Return value	Description
true	If the module synchronization is ongoing
false	If the module has completed synchronization

Function slcd_lock_shadow_memory()

Lock shadow memory.

```
void slcd_lock_shadow_memory( void )
```

It allows update of shadow display memory. If the display memory is modified, the display remains unchanged when locked.

Function slcd_unlock_shadow_memory()

Unlock shadow memory.

```
void slcd_unlock_shadow_memory( void )
```

Unlock the shadow display memory.

Function slcd_clear_display_memory()

Clear display memory.

```
void slcd_clear_display_memory( void )
```

Clears immediately the display memory.

Function slcd_enable_display()

Display enable.

```
void slcd_enable_display( void )
```

Enable COM/SEG signal output.

Function slcd_disable_display()

Display disable.

```
void slcd_disable_display( void )
```

Disable COM/SEG signal output.

Function slcd_dma_display_memory_update_fc_sel()

DMA display memory update frame counter selection.

```
void slcd_dma_display_memory_update_fc_sel(
    enum slcd_frame_counter fc)
```

It's used to select the frame counter for DMA to update the display memory.

Note: It can be called only before the module is enabled.

Table 20-13. Parameters

Data direction	Parameter name	Description
[in]	fc	Frame councter index

20.6.4.2. SLCD Blink Functions

Function `slcd_enable_blink()`

Blink mode enable.

```
void slcd_enable_blink( void )
```

Enable blink mode.

Function `slcd_disable_blink()`

Blink mode disable.

```
void slcd_disable_blink( void )
```

Disable blink mode.

Function `slcd_blink_get_config_defaults()`

Initializes SLCD blink configurations struct to defaults.

```
void slcd_blink_get_config_defaults(
    struct slcd_blink_config * blink_config)
```

Initailizes SLCD blink configuration struct to predefined safe default settings.

Table 20-14. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an instance of struct <code>slcd_blink_config</code>

Function `slcd_blink_set_config()`

Set SLCD blink mode.

```
enum status_code slcd_blink_set_config(
    struct slcd_blink_config *const blink_config)
```

Set SLCD blink mode.

Note: SLCD blink cannot be set while module or blink is enabled.

Table 20-15. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an SLCD blink configuration structure

Returns

Status of the configuration procedure.

Table 20-16. Return Values

Return value	Description
STATUS_OK	SLCD blink configuration went successful
STATUS_ERR_INVALID_ARG	If blink configuration failed

Function slcd_clear_blink_all_pixel()

Stop all SLCD pixels/segments from blinking.

```
void slcd_clear_blink_all_pixel( void )
```

Function slcd_clear_blink_pixel()

Stop a specified SLCD pixel/segment from blinking.

```
void slcd_clear_blink_pixel(
    uint8_t pix_com,
    uint8_t pix_seg)
```

Table 20-17. Parameters

Data direction	Parameter name	Description
[in]	pix_com	Pixel/segment COM coordinate
[in]	pix_seg	Pixel/segment SEG coordinate (range 0 to 1 inclusive)

Function slcd_set_blink_pixel()

Start an SLCD pixel/segment blinking.

```
void slcd_set_blink_pixel(
    uint8_t pix_com,
    uint8_t pix_seg)
```

Table 20-18. Parameters

Data direction	Parameter name	Description
[in]	pix_com	Pixel/segment COM coordinate
[in]	pix_seg	Pixel/segment SEG coordinate (range 0 to 1 inclusive)

20.6.4.3. SLCD Blank Functions

Function slcd_enable_blank()

Blank mode enable.

```
void slcd_enable_blank( void )
```

Enable blank mode.

Function slcd_disable_blank()

Blank mode disable.

```
void slcd_disable_blank( void )
```

Disable blank mode.

20.6.4.4. SLCD Event Functions

Function `slcd_enable_events()`

Enables a SLCD event output.

```
void slcd_enable_events(  
    struct slcd_events *const events)
```

Enables one or more output events.

Note: Events cannot be altered while the module is enabled.

Table 20-19. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to enable

Function `slcd_disable_events()`

Disables a SLCD event output.

```
void slcd_disable_events(  
    struct slcd_events *const events)
```

Disables one or more SLCD events output.

Table 20-20. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to disable

20.6.4.5. SLCD Frame Counter Functions

Function `slcd_set_frame_counter()`

Frame counter configuration.

```
void slcd_set_frame_counter(  
    enum slcd_frame_counter fc,  
    bool presc_bypass_enable,  
    uint16_t overflow_value)
```

Config frame counter.

Note: Frame counter cannot be set while it is enabled.

Table 20-21. Parameters

Data direction	Parameter name	Description
[in]	fc	Frame counter index
[in]	presc_bypass_enable	Bypass of the frame counter prescaler
[in]	overflow_value	Frame counter overflow value. The number of frame before overflow is ((overflow_value+1)*8) when presc_bypass_enable=0 else (overflow_value+1). The MAX. overflow value is 0x1FFFF.

Function slcd_enable_frame_counter()

Enables a frame counter.

```
void slcd_enable_frame_counter(
    enum slcd_frame_counter fc)
```

Enables one frame counter.

Table 20-22. Parameters

Data direction	Parameter name	Description
[in]	fc	Frame counter index

Function slcd_disable_frame_counter()

Disable a frame counter.

```
void slcd_disable_frame_counter(
    enum slcd_frame_counter fc)
```

Disable one frame counter.

Table 20-23. Parameters

Data direction	Parameter name	Description
[in]	fc	Frame counter index

20.6.4.6. Display Memory Functions

CPU can access display memory in direct access or in indirect access.

Function slcd_set_display_memory()

Set all bits in the SLCD display memory high.

```
void slcd_set_display_memory( void )
```

Function slcd_set_pixel()

Enable the specified pixel/segment in the SLCD display memory.

```
void slcd_set_pixel(
    uint8_t pix_com,
    uint8_t pix_seg)
```

Table 20-24. Parameters

Data direction	Parameter name	Description
[in]	pix_com	Pixel/segment COM coordinate, within [0-7]
[in]	pix_seg	Pixel/segment SEG coordinate within [0-43]

Function slcd_clear_pixel()

Disable the specified pixel/segment in the SLCD display memory.

```
void slcd_clear_pixel(
    uint8_t pix_com,
    uint8_t pix_seg)
```

Table 20-25. Parameters

Data direction	Parameter name	Description
[in]	pix_com	Pixel/segment COM coordinate
[in]	pix_seg	Pixel/segment SEG coordinate

Function slcd_set_seg_data()

Set the specified segment in the SLCD display memory.

```
void slcd_set_seg_data(
    uint8_t seg_data,
    uint8_t byte_offset,
    uint8_t seg_mask)
```

Table 20-26. Parameters

Data direction	Parameter name	Description
[in]	pix_seg	Pixel/segment SEG coordinate
[in]	byte_offset	Byte offset in display memory
[in]	seg_mask	Byte offset in display memory

20.6.4.7. Character Mapping Functions**Function slcd_character_map_set()**

Set SLCD character mapping.

```
void slcd_character_map_set(
    enum slcd_automated_char_order order,
    uint8_t seg_line_num)
```

Set Character mode and SEG line per digit.

Table 20-27. Parameters

Data direction	Parameter name	Description
[in]	order	Mapping order in char mode
[in]	seg_line_num	Define the number of SEG line per digit, it equal to number of SEG line - 1

Function slcd_character_write_data()

Write segments data to display memory in character mode.

```
void slcd_character_write_data(
    uint8_t com_line_index,
    uint8_t seg_line_index,
    uint32_t seg_data,
    uint32_t data_mask)
```

Table 20-28. Parameters

Data direction	Parameter name	Description
[in]	seg_data	Pixel/segment data
[in]	data_mask	Segments data mask
[in]	com_line_index	COM line index
[in]	seg_line_index	Segments line index

Function slcd_enable_automated_character()

Enables automated character display.

```
void slcd_enable_automated_character( void )
```

Enables automated character display.

Function slcd_disable_automated_character()

Disables automated character display.

```
void slcd_disable_automated_character( void )
```

Disables automated character display.

Function slcd_automated_char_get_config_default()

Initializes SLCD Automated Character configurations struct to defaults.

```
void slcd_automated_char_get_config_default(
    struct slcd_automated_char_config * config)
```

Initializes SLCD Automated Character configuration struct to predefined safe default settings.

Table 20-29. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an instance of struct slcd_automated_char_config

Function `slcd_automated_char_set_config()`

Set SLCD automated character.

```
enum status_code slcd_automated_char_set_config(  
    struct slcd_automated_char_config *const config)
```

Set automated character mode.

Note: SLCD automated character mode cannot be set while module or automated character is enabled.

Table 20-30. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an SLCD automated character configuration structure

Returns

Status of the configuration procedure.

Table 20-31. Return Values

Return value	Description
STATUS_OK	SLCD automated character configuration went successful
STATUS_ERR_INVALID_ARG	If automated character configuration failed

20.6.4.8. Automated Bit Mapping Functions

Function `slcd_enable_automated_bit()`

Enables automated bit display.

```
void slcd_enable_automated_bit( void )
```

Enables automated bit display.

Function `slcd_disable_automated_bit()`

Disables automated bit display.

```
void slcd_disable_automated_bit( void )
```

Disables automated bit display.

Function `slcd_set_automated_bit()`

Sets automated bit display.

```
void slcd_set_automated_bit(  
    uint8_t size,  
    enum slcd_frame_counter fc)
```

Sets automated bit display.

Note: Automated bit cannot be set while it is enabled or busy.

20.6.4.9. Autonomous Segment Animation

Function `slcd_enable_circular_shift()`

Enable SLCD circular shift mode.

```
void slcd_enable_circular_shift( void )
```

Function `slcd_disable_circular_shift()`

Disable SLCD circular shift mode.

```
void slcd_disable_circular_shift( void )
```

Function `slcd_circular_shift_get_config_defaults()`

Initializes circular shift configurations struct to defaults.

```
void slcd_circular_shift_get_config_defaults(  
    struct slcd_circular_shift_config *const config)
```

Initializes circular shift configuration struct to predefined safe default settings.

Table 20-32. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an instance of struct <code>slcd_circular_shift_config</code>

Function `slcd_circular_shift_set_config()`

Set SLCD circular shift.

```
enum status_code slcd_circular_shift_set_config(  
    struct slcd_circular_shift_config *const config)
```

Set circular shift mode.

Note: SLCD circular shift mode cannot be set while module or circular shift is enabled.

Table 20-33. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an SLCD circular shift configuration structure

Returns

Status of the configuration procedure.

Table 20-34. Return Values

Return value	Description
STATUS_OK	SLCD circular shift configuration went successful
STATUS_ERR_INVALID_ARG	If circular shift configuration failed

20.6.4.10. SLCD Status

Function `slcd_get_auto_bit_status()`

Checks if auto bit mapping state machine is busy.

```
bool slcd_get_auto_bit_status( void )
```

Checks if auto bit mapping state machine is busy or not.

Table 20-35. Return Values

Return value	Description
true	Auto bit mapping state machine is busy
false	Auto bit mapping state machine is idle

Function `slcd_get_auto_char_status()`

Checks if auto character mapping state machine is busy.

```
bool slcd_get_auto_char_status( void )
```

Checks if auto character state machine is busy or not.

Table 20-36. Return Values

Return value	Description
true	Auto character mapping state machine is busy
false	Auto character mapping state machine is idle

Function `slcd_get_char_writing_status()`

Checks if character writing function is busy.

```
bool slcd_get_char_writing_status( void )
```

Checks if character writing function is busy or not.

Table 20-37. Return Values

Return value	Description
true	Character writing function is busy
false	Character writing function is ready for use

Function `slcd_get_vlcd_vdd33_status()`

Checks VLCD and VDD33 status.

```
bool slcd_get_vlcd_vdd33_status( void )
```

Checks VLCD and VDD33 status.

Table 20-38. Return Values

Return value	Description
true	VDD33 is greater than target VLCD
false	Target VLCD is greater than vdd33

Function slcd_get_charge_pump_status()

Checks LCD charge pump status.

```
bool slcd_get_charge_pump_status( void )
```

Checks LCD Charge Pump Status.

Table 20-39. Return Values

Return value	Description
true	LCD power charge pump is running
false	LCD power charge pump is stopped

Function slcd_get_vlcd_ready_status()

Checks if VLCD is ready.

```
bool slcd_get_vlcd_ready_status( void )
```

Checks if VLCD is well regulated to the target value.

Table 20-40. Return Values

Return value	Description
true	VLCD is well regulated to the target value
false	VLCD is not well regulated to the target value

20.6.4.11. Callback Function**Function slcd_register_callback()**

Registers a callback.

```
enum status_code slcd_register_callback(
    const slcd_callback_t callback,
    const enum slcd_callback_type type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled by [slcd_enable_callback](#), in order for the interrupt handler to call it when the conditions for the callback type is met.

Table 20-41. Parameters

Data direction	Parameter name	Description
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

Table 20-42. Return Values

Return value	Description
STATUS_OK	The function exited successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function slcd_unregister_callback()

Unregisters a callback.

```
enum status_code slcd_unregister_callback(
    const slcd_callback_t callback,
    const enum slcd_callback_type type)
```

Unregisters a callback function implemented by the user.

Table 20-43. Parameters

Data direction	Parameter name	Description
[in]	callback_type	Callback type given by an enum

Table 20-44. Return Values

Return value	Description
STATUS_OK	The function exited successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function slcd_enable_callback()

Enable an SLCD callback.

```
void slcd_enable_callback(
    const enum slcd_callback_type type)
```

Table 20-45. Parameters

Data direction	Parameter name	Description
[in]	type	Callback source type

Table 20-46. Return Values

Return value	Description
STATUS_OK	The function exited successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function slcd_disable_callback()

Disable an SLCD callback.

```
void slcd_disable_callback(
    const enum slcd_callback_type type)
```

Table 20-47. Parameters

Data direction	Parameter name	Description
[in]	type	Callback source type

Table 20-48. Return Values

Return value	Description
STATUS_OK	The function exited successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

20.6.5. Enumeration Definitions**20.6.5.1. Enum slcd_automated_char_mode**

Enum automated char display mode.

Table 20-49. Members

Enum value	Description
SLCD_AUTOMATED_CHAR_SEQ	Sequential Display Mode
SLCD_AUTOMATED_CHAR_SCROLL	Scrolling Display Mode

20.6.5.2. Enum slcd_automated_char_order

Enum automated char order.

Table 20-50. Members

Enum value	Description
SLCD_AUTOMATED_CHAR_START_FROM_BOTTOM_RIGHT	Segment is starting from bottom right
SLCD_AUTOMATED_CHAR_START_FROM_BOTTOM_LEFT	Segment is starting from bottom left

20.6.5.3. Enum slcd_callback_type

Enum SLCD callback type.

Table 20-51. Members

Enum value	Description
SLCD_CALLBACK_FC0_OVERFLOW	Frame Counter 0 Overflow callback
SLCD_CALLBACK_FC1_OVERFLOW	Frame Counter 1 Overflow callback
SLCD_CALLBACK_FC2_OVERFLOW	Frame Counter 2 Overflow callback
SLCD_CALLBACK_VLCD_READY	VLCD Ready Toggle callback
SLCD_CALLBACK_VLCD_TOGGLE	VLCD Status Toggle callback
SLCD_CALLBACK_PUMP_TOGGLE	Pump Run Status Toggle callback

20.6.5.4. Enum slcd_circular_shift_dir

Enum SLCD circular shift direction.

Table 20-52. Members

Enum value	Description
SLCD_CIRCULAR_SHIFT_LEFT	Circular shift direction is left
SLCD_CIRCULAR_SHIFT_RIGHT	Circular shift direction is right

20.6.5.5. Enum slcd_frame_counter

Enum SLCD frame counter definition.

Table 20-53. Members

Enum value	Description
SLCD_FRAME_COUNTER_0	SLCD frame counter 0
SLCD_FRAME_COUNTER_1	SLCD frame counter 1
SLCD_FRAME_COUNTER_2	SLCD frame counter 2

20.6.5.6. Enum slcd_waveform_mode

Enum waveform mode.

Table 20-54. Members

Enum value	Description
SLCD_LOW_POWER_WAVEFORM_MODE	Low power waveform mode
SLCD_STANDARD_WAVEFORM_MODE	Standard waveform mode

20.7. Extra Information for SLCD

20.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Definition
SLCD	Segment Liquid Crystal Display
COM	Common, denotes how many segments are connected to a segment terminal
SEG	Segment, the least viewing element (pixel) which can be on or off
Duty	$1/(\text{Number of common terminals on an actual LCD display})$
Bias	$1/(\text{Number of voltage levels used driving a LCD display} - 1)$
Frame Rate	Number of times the LCD segments are energized per second

20.7.2. Dependencies

This driver has the following dependencies:

- None

20.7.3. Errata

There are no errata related to this driver.

20.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial release

20.8. Examples for SLCD

This is a list of the available Quick Start Guides (QSGs) and example applications for [SAM Segment Liquid Crystal Display\(SLCD\) Controller](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for SLCD](#)

20.8.1. Quick Start Guide for SLCD

The supported board list:

- SAM L22 Xplained Pro

The SEGMENT LCD1 Xplained Pro extension board must be connected to extension header 5 on the SAM L22 Xplained Pro.

This example demonstrates how to use the SLCD driver, it covers the following cases:

- Display Memory Mapping(Direct Access and Indirect Access)
- Character Mapping
- Blinking
- Automated Character Mapping
- Automated Bit Mapping

Upon startup, the program uses the USART driver to display application output message.

20.8.1.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, outside of any functions:

```
/*Character map 0-9,A-Z*/
const uint32_t character_map[] = {
    0x2e74,0x440,0x23c4,0x25c4,0x5e0,0x25a4,0x27a4,0x444,0x27e4,0x25e4, /
    *0-9*/
    0x7e4,0xa545,0x2224,0xa445,0x23a4,0x3a4,0x2724, /*A-G*/
    0x7e0,0xa005,0x2640,0x12b0,0x2220,0x678,0x1668, /*H-N*/
    0x2664,0x3e4,0x3664,0x13e4,0x25a4,0x8005, /*O-T*/
    0x2660,0xa30,0x1e60,0x1818,0x8018,0x2814 /*U-Z*/
};

/* HELLO ATMEL map */
uint32_t display_array[11]={0x7e0,0x23a4,0x2220,0x2220,0x2664,0,
    0x7e4,0x8005,0x678,0x23a4,0x2220};
#define DISPLAY_ARRAY_SIZE (sizeof(display_array)/sizeof(uint32_t))

/*ICON COM/SEG map */
#define C42412A_ICON_USB          1, 1
#define C42412A_ICON_BAT          0, 0
#define C42412A_ICON_ATMEL        0, 1

/* Automated Bit Mapping
Format: [00,offset,data mask,data]
data:[7-0]
data mask:[15-8]
offset:[21-16]
*/
const uint32_t abm_matrix_string0[] = {
    0x00002f10,0x00012211,0x00022211, /* COM0 */
    0x00060f00,0x00070000,0x00080000, /* COM1 */
    0x000c0f80,0x000d0088,0x000e0088, /* COM2 */
    0x00124f10,0x00134411,0x00144411 /* COM3 */
};

const uint32_t abm_matrix_string1[] = {
    0x00002f10,0x00012211,0x00022211, /* COM0 */
    0x00060f80,0x00070088,0x00080088, /* COM1 */
    0x000c0f90,0x000d0099,0x000e0099, /* COM2 */
    0x00124f10,0x00134411,0x00144411 /* COM3 */
};

const uint32_t abm_matrix_string2[] = {
```

```

    0x00002f90, 0x00012299, 0x00022299, /* COM0 */
    0x00060f10, 0x00070011, 0x00080011, /* COM1 */
    0x000c0f80, 0x000d0088, 0x000e0088, /* COM2 */
    0x00124f10, 0x00134411, 0x00144411 /* COM3 */
};

const uint32_t abm_matrix_string3[] = {
    0x00002f10, 0x00012211, 0x00022211, /* COM0 */
    0x00060f00, 0x00070000, 0x00080000, /* COM1 */
    0x000c0f00, 0x000d0000, 0x000e0000, /* COM2 */
    0x00124f80, 0x00134488, 0x00144488 /* COM3 */
};

#define ABM_MATRIX_SIZE (sizeof(abm_matrix_string0)/sizeof(uint32_t))

```

Add to the main application source file, outside of any functions:

```

struct dma_resource example_resource_abm;
struct dma_resource example_resource_acm;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_acm ;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm1;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm2;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm3;

struct usart_module usart_instance;

```

Copy-paste the following setup code to your user application:

```

static void configure_console(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}

static void configure_dma_acm(void)
{
    struct dma_resource_config acm_config;

    dma_get_config_defaults(&acm_config);

    acm_config.peripheral_trigger = SLCD_DMAL_ID_ACMDRDY;
    acm_config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(&example_resource_acm, &acm_config);

    struct dma_descriptor_config acm_descriptor_config;

    dma_descriptor_get_config_defaults(&acm_descriptor_config);

    acm_descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
}

```



```

    acm_descriptor_config.src_increment_enable = true;
    acm_descriptor_config.block_transfer_count = DISPLAY_ARRAY_SIZE;
    acm_descriptor_config.step_selection = DMA_STEPSEL_SRC;
    acm_descriptor_config.dst_increment_enable = false;
    acm_descriptor_config.source_address = (uint32_t)display_array
                                          + sizeof(display_array);
    acm_descriptor_config.destination_address = (uint32_t)&SLCD-
>CMDATA.reg;
    acm_descriptor_config.next_descriptor_address =
    (uint32_t)&example_descriptor_acm;

    dma_descriptor_create(&example_descriptor_acm, &acm_descriptor_config);

    dma_add_descriptor(&example_resource_acm, &example_descriptor_acm);
    dma_start_transfer_job(&example_resource_acm);
}

static void configure_dma_abm(void)
{
    struct dma_resource_config abm_config;
    dma_get_config_defaults(&abm_config);
    abm_config.peripheral_trigger = SLCD_DMACH_ID_ABMDRDY;
    abm_config.trigger_action = DMA_TRIGGER_ACTION_BEAT;

    dma_allocate(&example_resource_abm, &abm_config);

    struct dma_descriptor_config abm_descriptor_config;
    dma_descriptor_get_config_defaults(&abm_descriptor_config);

    abm_descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
    abm_descriptor_config.src_increment_enable = true;
    abm_descriptor_config.step_selection = DMA_STEPSEL_SRC;
    abm_descriptor_config.dst_increment_enable = false;
    abm_descriptor_config.block_transfer_count = ABM_MATRIX_SIZE;
    abm_descriptor_config.source_address = (uint32_t)abm_matrix_string0
                                          + sizeof(abm_matrix_string0);
    abm_descriptor_config.destination_address = (uint32_t)&SLCD-
>ISDATA.reg;
    abm_descriptor_config.next_descriptor_address =
    (uint32_t)&example_descriptor_abm1;

    dma_descriptor_create(&example_descriptor_abm, &abm_descriptor_config);

    abm_descriptor_config.source_address = (uint32_t)abm_matrix_string1
                                          + sizeof(abm_matrix_string1);
    abm_descriptor_config.next_descriptor_address =
    (uint32_t)&example_descriptor_abm2;
    dma_descriptor_create(&example_descriptor_abm1,
    &abm_descriptor_config);

    abm_descriptor_config.source_address = (uint32_t)abm_matrix_string2
                                          + sizeof(abm_matrix_string2);
    abm_descriptor_config.next_descriptor_address =
    (uint32_t)&example_descriptor_abm3;
    dma_descriptor_create(&example_descriptor_abm2,
    &abm_descriptor_config);

    abm_descriptor_config.source_address = (uint32_t)abm_matrix_string3
                                          + sizeof(abm_matrix_string3);
    abm_descriptor_config.next_descriptor_address =
    (uint32_t)&example_descriptor_abm;
    dma_descriptor_create(&example_descriptor_abm3,

```

```

&abm_descriptor_config);

    dma_add_descriptor(&example_resource_abm, &example_descriptor_abm);
    dma_start_transfer_job(&example_resource_abm);
}

```

Add to user application initialization (typically the start of `main()`):

```

struct slcd_config config;
system_init();
configure_console();
delay_init();

/* Turn on the backlight. */
port_pin_set_output_level(SLCD_BACKLIGHT,true);

printf("SLCD example starts\r\n");
slcd_get_config_defaults(&config);
slcd_init(&config);
slcd_set_contrast(0x8);
configure_dma_acm();
configure_dma_abm();

slcd_enable();

```

Workflow

1. Define character map data.

```

/*Character map 0-9,A-Z*/
const uint32_t character_map[] = {

0x2e74,0x440,0x23c4,0x25c4,0x5e0,0x25a4,0x27a4,0x444,0x27e4,0x25e4, /
/*0-9*/
    0x7e4,0xa545,0x2224,0xa445,0x23a4,0x3a4,0x2724, /*A-G*/
    0x7e0,0xa005,0x2640,0x12b0,0x2220,0x678,0x1668, /*H-N*/
    0x2664,0x3e4,0x3664,0x13e4,0x25a4,0x8005,/*O-T*/
    0x2660,0xa30,0x1e60,0x1818,0x8018,0x2814/*U-Z*/
};

/* HELLO ATMELO map */
uint32_t display_array[11]={0x7e0,0x23a4,0x2220,0x2220,0x2664,0,
                           0x7e4,0x8005,0x678,0x23a4,0x2220};
#define DISPLAY_ARRAY_SIZE (sizeof(display_array)/sizeof(uint32_t))

/*ICON COM/SEG map */
#define C42412A_ICON_USB          1, 1
#define C42412A_ICON_BAT          0, 0
#define C42412A_ICON_ATMEL        0, 1

/* Automated Bit Mapping
Format: [00,offset,data mask,data]
data:[7-0]
data mask:[15-8]
offset:[21-16]
*/
const uint32_t abm_matrix_string0[] = {
    0x00002f10,0x00012211,0x00022211, /* COM0 */
    0x00060f00,0x00070000,0x00080000, /* COM1 */
    0x000c0f80,0x000d0088,0x000e0088, /* COM2 */
    0x00124f10,0x00134411,0x00144411 /* COM3 */
};

```

```

const uint32_t abm_matrix_string1[] = {
    0x00002f10, 0x00012211, 0x00022211, /* COM0 */
    0x00060f80, 0x00070088, 0x00080088, /* COM1 */
    0x000c0f90, 0x000d0099, 0x000e0099, /* COM2 */
    0x00124f10, 0x00134411, 0x00144411 /* COM3 */
};

const uint32_t abm_matrix_string2[] = {
    0x00002f90, 0x00012299, 0x00022299, /* COM0 */
    0x00060f10, 0x00070011, 0x00080011, /* COM1 */
    0x000c0f80, 0x000d0088, 0x000e0088, /* COM2 */
    0x00124f10, 0x00134411, 0x00144411 /* COM3 */
};

const uint32_t abm_matrix_string3[] = {
    0x00002f10, 0x00012211, 0x00022211, /* COM0 */
    0x00060f00, 0x00070000, 0x00080000, /* COM1 */
    0x000c0f00, 0x000d0000, 0x000e0000, /* COM2 */
    0x00124f80, 0x00134488, 0x00144488 /* COM3 */
};

#define ABM_MATRIX_SIZE (sizeof(abm_matrix_string0)/sizeof(uint32_t))

```

2. Create related module variable and software instance structure.

```

struct dma_resource example_resource_abm;
struct dma_resource example_resource_acm;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_acm ;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm1;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm2;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_abm3;

struct usart_module usart_instance;

```

3. Configure, initialize, and enable slcd module.

1. Configuration slcd struct, which can be filled out to adjust the configuration of a physical slcd peripheral.

```

slcd_get_config_defaults(&config);
slcd_init(&config);
slcd_set_contrast(0x8);

```

2. Config DMA module for automated character mapping and automated bit mapping.

```

configure_dma_acm();
configure_dma_abm();

```

3. Enable the slcd module.

```

slcd_enable();

```

20.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
/* 1. Display all*/
slcd_set_display_memory();
delay_s(1);
slcd_clear_display_memory();

/* 2. Display icon*/
slcd_set_pixel(C42412A_ICON_USB);
slcd_set_pixel(C42412A_ICON_BAT);
slcd_set_pixel(C42412A_ICON_ATMEL);
delay_s(1);

/* 3. Character map*/
slcd_character_map_set(SLCD_AUTOMATED_CHAR_START_FROM_BOTTOM_RIGHT, 3);
for(uint32_t i = 0; i < 5; i++) {
    slcd_character_write_data(0, 4+i*4, character_map[10+i], 0xFF4002);
}
delay_s(2);

/* 4. Blinking*/
slcd_disable();
struct slcd_blink_config blink_config;
slcd_blink_get_config_defaults(&blink_config);
blink_config.blink_all_seg = false;
slcd_blink_set_config(&blink_config);

for(uint32_t i=0; i<4; i++){
    slcd_set_blink_pixel(i, 0);
    slcd_set_blink_pixel(i, 1);
}

slcd_set_frame_counter(SLCD_FRAME_COUNTER_0, true, 0x18);
slcd_enable_frame_counter(SLCD_FRAME_COUNTER_0);

slcd_enable_blink();
slcd_enable();
delay_s(2);

/* 5. Automated Character Mapping*/
slcd_disable();
struct slcd_automated_char_config acm_config;
slcd_automated_char_get_config_default(&acm_config);
acm_config.order = SLCD_AUTOMATED_CHAR_START_FROM_BOTTOM_RIGHT;
acm_config.fc = SLCD_FRAME_COUNTER_1;
acm_config.mode = SLCD_AUTOMATED_CHAR_SCROLL;
acm_config.seg_line_num = 3;
acm_config.start_seg_line = 4;
acm_config.row_digit_num = 5;
acm_config.digit_num = 5;
acm_config.scrolling_step = sizeof(display_array) - 5 + 1;
acm_config.com_line_num = 4;
acm_config.data_mask = 0xFF4002;
slcd_automated_char_set_config(&acm_config);

slcd_set_frame_counter(SLCD_FRAME_COUNTER_1, true, 0x1E);
slcd_enable_frame_counter(SLCD_FRAME_COUNTER_1);

slcd_enable_automated_character();
slcd_enable();
```

```

delay_s(3);

/* 6. Automated Bit Mapping*/
slcd_disable();
slcd_disable_automated_character();
slcd_clear_display_memory();
slcd_disable_frame_counter(SLCD_FRAME_COUNTER_2);
slcd_set_automated_bit(ABM_MATRIX_SIZE,SLCD_FRAME_COUNTER_2);
slcd_set_frame_counter(SLCD_FRAME_COUNTER_2,true,0x1F);
slcd_enable_frame_counter(SLCD_FRAME_COUNTER_2);
slcd_enable_automated_bit();
slcd_enable();

```

Workflow

1. Display all case.

```

/* 1. Display all*/
slcd_set_display_memory();
delay_s(1);
slcd_clear_display_memory();

```

2. Display icon.

```

/* 2. Display icon*/
slcd_set_pixel(C42412A_ICON_USB);
slcd_set_pixel(C42412A_ICON_BAT);
slcd_set_pixel(C42412A_ICON_ATMEL);
delay_s(1);

```

3. Character map case.

```

/* 3. Character map*/
slcd_character_map_set(SLCD_AUTOMATED_CHAR_START_FROM_BOTTOM_RIGHT,3);
for(uint32_t i = 0 ; i < 5 ; i++) {
    slcd_character_write_data(0,4+i*4,character_map[10+i],0xFF4002);
}
delay_s(2);

```

4. SLCD blinking.

```

/* 4. Blinking*/
slcd_disable();
struct slcd_blink_config blink_config;
slcd_blink_get_config_defaults(&blink_config);
blink_config.blink_all_seg = false;
slcd_blink_set_config(&blink_config);

for(uint32_t i=0; i<4; i++){
    slcd_set_blink_pixel(i,0);
    slcd_set_blink_pixel(i,1);
}

slcd_set_frame_counter(SLCD_FRAME_COUNTER_0,true,0x18);
slcd_enable_frame_counter(SLCD_FRAME_COUNTER_0);

slcd_enable_blink();
slcd_enable();
delay_s(2);

```

5. Automated character mapping.

```

/* 5. Automated Character Mapping*/
slcd_disable();
struct slcd_automated_char_config acm_config;

```

```

slcd_automated_char_get_config_default(&acm_config);
acm_config.order = SLCD_AUTOMATED_CHAR_START_FROM_BOTTOM_RIGHT;
acm_config.fc = SLCD_FRAME_COUNTER_1;
acm_config.mode = SLCD_AUTOMATED_CHAR_SCROLL;
acm_config.seg_line_num = 3;
acm_config.start_seg_line = 4;
acm_config.row_digit_num = 5;
acm_config.digit_num = 5;
acm_config.scrolling_step = sizeof(display_array) - 5 + 1;
acm_config.com_line_num = 4;
acm_config.data_mask = 0xFF4002;
slcd_automated_char_set_config(&acm_config);

slcd_set_frame_counter(SLCD_FRAME_COUNTER_1, true, 0x1E);
slcd_enable_frame_counter(SLCD_FRAME_COUNTER_1);

slcd_enable_automated_character();
slcd_enable();
delay_s(3);

```

6. Automated bit mapping.

```

/* 6. Automated Bit Mapping*/
slcd_disable();
slcd_disable_automated_character();
slcd_clear_display_memory();
slcd_disable_frame_counter(SLCD_FRAME_COUNTER_2);
slcd_set_automated_bit(ABM_MATRIX_SIZE, SLCD_FRAME_COUNTER_2);
slcd_set_frame_counter(SLCD_FRAME_COUNTER_2, true, 0x1F);
slcd_enable_frame_counter(SLCD_FRAME_COUNTER_2);
slcd_enable_automated_bit();
slcd_enable();

```

21. SAM System (SYSTEM) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's system relation functionality, necessary for the basic device operation. This is not limited to a single peripheral, but extends across multiple hardware peripherals.

The following peripherals are used by this module:

- PM (Power Manager)
- RSTC (Reset Controller)
- SUPC (Supply Controller)

The following devices can use this module:

- Atmel | SMART SAM L21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

21.1. Prerequisites

There are no prerequisites for this module.

21.2. Module Overview

The System driver provides a collection of interfaces between the user application logic, and the core device functionality (such as clocks, reset cause determination, etc.) that is required for all applications. It contains a number of sub-modules that control one specific aspect of the device:

- System Core (this module)
- System Clock Control (sub-module)
- System Interrupt Control (sub-module)
- System Pin Multiplexer Control (sub-module)

21.2.1. Voltage Regulator

The SAM device controls the voltage regulators for the core (VDDCORE) and backup (VDDBU) domains. It sets the voltage regulators according to the sleep modes, the performance level, or the user configuration.

In active mode, the voltage regulator can be chosen on the fly between a LDO or a Buck converter. In standby mode, the low power voltage regulator is used to supply VDDCORE.

21.2.2. Battery Backup Power Switch

The SAM device supports connection of a battery backup to the VBAT power pin. It includes functionality that enables automatic power switching between main power and battery backup power. This will ensure power to the backup domain, when the main battery or power source is unavailable.

21.2.3. Voltage References

The various analog modules within the SAM devices (such as AC, ADC, and DAC) require a voltage reference to be configured to act as a reference point for comparisons and conversions.

The SAM devices contain multiple references, including an internal temperature sensor and a fixed band-gap voltage source. When enabled, the associated voltage reference can be selected within the desired peripheral where applicable.

21.2.4. System Reset Cause

In some applications there may be a need to execute a different program flow based on how the device was reset. For example, if the cause of reset was the Watchdog timer (WDT), this might indicate an error in the application, and a form of error handling or error logging might be needed.

For this reason, an API is provided to retrieve the cause of the last system reset, so that appropriate action can be taken.

There are three groups of reset sources:

- Power supply reset: Resets caused by an electrical issue. It covers POR and BOD reset.
- User reset: Resets caused by the application. It covers external reset, system reset, and watchdog reset.
- Backup reset: Resets caused by a backup mode exit condition.

21.2.5. Performance Level

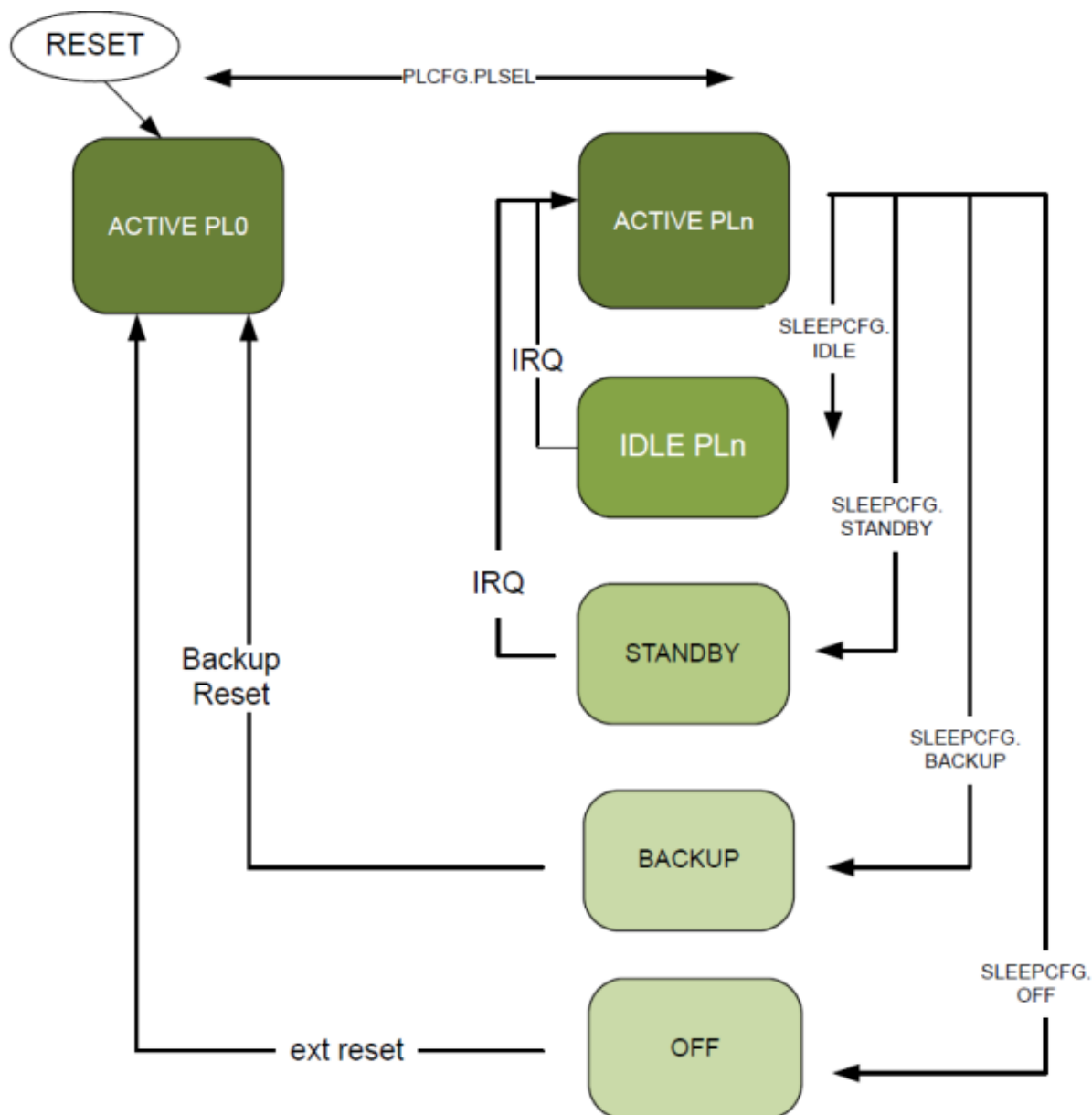
Performance level allows the user to adjust the regulator output voltage to reduce power consumption. The user can on the fly select the most suitable performance level, depending on the application demands.

The SAM device can operate at two different performance levels (PL0 and PL2). When operating at PL0, the voltage applied on the full logic area is reduced by voltage scaling. This voltage scaling technique allows to reduce the active power consumption while decreasing the maximum frequency of the device. When operating at PL2, the voltage regulator supplies the highest voltage, allowing the device to run at higher clock speeds.

Performance level transition is possible only when the device is in active mode. After a reset, the device starts at the lowest performance level (lowest power consumption and lowest max. frequency). The application can then switch to another performance level at any time without any stop in the code execution. As shown in [Figure 21-1](#).

Note: When scaling down the performance level, the bus frequency should first be scaled down in order to not exceed the maximum frequency allowed for the low performance level. When scaling up the performance level (e.g. from PL0 to PL2), check the performance level status before increasing the bus frequency. It can be increased only when the performance level transition is completed.

Figure 21-1. Performance Level Transition



21.2.6. Power Domain Gating

Power domain gating allows power saving by reducing the voltage in logic areas in the device to a low-power supply. The feature is available in Standby sleep mode and will reduce the voltage in domains where all peripherals are idle. Internal logic will maintain its content, meaning the corresponding peripherals will not need to be reconfigured when normal operating voltage is returned. Most power domains can be in the following three states:

- Active state: The power domain is powered on.
- Retention state: The main voltage supply for the power domain is switched off, while maintaining a secondary low-power supply for the sequential cells. The logic context is restored when waking up.
- Off state: The power domain is entirely powered off. The logic context is lost.

The SAM L21 device contains three power domains which can be controlled using power domain gating, namely PD0, PD1, and PD2. These power domains can be configured to the following cases:

- Default with no sleepwalking peripherals: A power domain is automatically set to retention state in standby sleep mode if no activity require it. The application can force all power domains to remain in active state during standby sleep mode in order to accelerate wakeup time.
- Default with sleepwalking peripherals: If one or more peripherals are enabled to perform sleepwalking tasks in standby sleep mode, the corresponding power domain (PDn) remains in active state as well as all inferior power domains (<PDn).
- Sleepwalking with dynamic power domain gating: During standby sleep mode, a power domain (PDn) in active can wake up a superior power domain (>PDn) in order to perform a sleepwalking task. The superior power domain is then automatically set to active state. At the end of the sleepwalking task, the device can either be woken up or the superior power domain can return to retention state.

Power domains can be linked to each other, it allows a power domain (PDn) to be kept in active state if the inferior power domain (PDn-1) is in active state too.

Table 21-1 illustrates the four cases to consider in standby mode.

Table 21-1. Sleep Mode versus Power Domain State Overview

Sleep mode	PD0	PD1	PD2	PDTOP	PDBACKUP
Idle	active	active	active	active	active
Standby - Case 1	active	active	active	active	active
Standby - Case 2	active	active	retention	active	active
Standby - Case 3	active	retention	retention	active	active
Standby - Case 4	retention	retention	retention	active	active
Backup	off	off	off	off	active
Off	off	off	off	off	off

21.2.7. RAMs Low Power Mode

By default, in standby sleep mode, RAM is in low power mode (back biased) if its power domain is in retention state. Table 21-2 lists RAMs low power mode.

Table 21-2. RAM Back-biasing Mode

RAM mode	Description
Retention Back-biasing mode	RAM is back-biased if its power domain is in retention mode
Standby Back-biasing mode	RAM is back-biased if the device is in standby mode
Standby OFF mode	RAM is OFF if the device is in standby mode
Always OFF mode	RAM is OFF if the device is in RET mode

21.2.8. Sleep Modes

The SAM devices have several sleep modes. The sleep mode controls which clock systems on the device will remain enabled or disabled when the device enters a low power sleep mode. [Table 21-3](#) lists the clock settings of the different sleep modes.

Table 21-3. SAM Device Sleep Modes

Sleep mode	System clock	CPU clock	AHB/AHB clock	GCLK clocks	Oscillators (ONDEMAND = 0)	Oscillators (ONDEMAND = 1)	Regulator mode	RAM mode
Idle	Run	Stop	Run if requested	Run	Run	Run if requested	Normal	Normal
Standby	Stop	Stop	Run if requested	Run if requested	Run if requested or RUNSTDBY = 1	Run if requested	Low pwer	Low pwer
Backup	Stop	Stop	Stop	Stop	Stop	Stop	Backup	Off
Off	Off	Off	Off	Off	Off	Off	Off	Off

Before entering device sleep, one of the available sleep modes must be set. The device will automatically wake up in response to an interrupt being generated or upon any other sleep mode exit condition.

Some peripheral clocks will remain enabled during sleep, depending on their configuration. If desired, the modules can remain clocked during sleep to allow them continue to operate while other parts of the system are powered down to save power.

21.3. Special Considerations

Most of the functions in this driver have device specific restrictions and caveats; refer to your device datasheet.

21.4. Extra Information

For extra information, see [Extra Information for SYSTEM Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

21.5. Examples

For SYSTEM module related examples, refer to the sub-modules listed in the [Module Overview](#).

For a list of examples related to this driver, see [Examples for SYSTEM Driver](#).

21.6. API Overview

21.6.1. Structure Definitions

21.6.1.1. Struct `system_battery_backup_power_switch_config`

Configuration structure for Battery Backup Power Switch (BBPS).

Table 21-4. Members

Type	Name	Description
enum system_battery_power_switch	battery_power_switch	Battery backup power switch configuration
bool	wake_enabled	Enable device wake up when BBPS switches from battery backup power to main power

21.6.1.2. Struct `system_standby_config`

Configuration structure for standby mode.

Table 21-5. Members

Type	Name	Description
enum system_ram_back_bias_mode	hmcramchs_back_bias	Back bias for HMCRAMCHS.
enum system_ram_back_bias_mode	hmcramclp_back_bias	Back bias for HMCRAMCLP

21.6.1.3. Struct `system_voltage_references_config`

Configuration structure for VREF.

Table 21-6. Members

Type	Name	Description
bool	on_demand	On Demand Control
bool	run_in_standby	Run in standby
enum system_voltage_references_sel	sel	Voltage References Selection

21.6.1.4. Struct `system_voltage_regulator_config`

Configuration structure for VREG.

Table 21-7. Members

Type	Name	Description
enum system_voltage_regulator_low_power_efficiency	low_power_efficiency	Low power efficiency
enum system_voltage_regulator_sel	regulator_sel	Voltage Regulator Selection

Type	Name	Description
bool	run_in_standby	Run in standby in standby sleep mode
uint8_t	voltage_scale_period	Voltage scaling period
uint8_t	voltage_scale_step	Voltage scaling voltage step

21.6.2. Function Definitions

21.6.2.1. Voltage Regulator

Function `system_voltage_regulator_get_config_defaults()`

Retrieve the default configuration for voltage regulator.

```
void system_voltage_regulator_get_config_defaults(
    struct system_voltage_regulator_config *const config)
```

Fills a configuration structure with the default configuration:

- Voltage scaling period is 1µs
- Voltage scaling voltage step is 2*min_step
- The voltage regulator is in low power mode in Standby sleep mode
- The voltage regulator in active mode is an LDO voltage regulator
- The voltage regulator in Low power mode has the default efficiency

Table 21-8. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_voltage_regulator_set_config()`

Configure voltage regulator.

```
void system_voltage_regulator_set_config(
    struct system_voltage_regulator_config *const config)
```

Configures voltage regulator with the given configuration.

Table 21-9. Parameters

Data direction	Parameter name	Description
[in]	config	Voltage regulator configuration structure containing the new config

Function `system_voltage_regulator_enable()`

Enable the selected voltage regulator.

```
void system_voltage_regulator_enable( void )
```

Enables the selected voltage regulator source.

Function `system_voltage_regulator_disable()`

Disable the selected voltage regulator.

```
void system_voltage_regulator_disable( void )
```

Disables the selected voltage regulator.

21.6.2.2. Voltage References

Function `system_voltage_reference_get_config_defaults()`

Retrieve the default configuration for voltage reference.

```
void system_voltage_reference_get_config_defaults(  
    struct system_voltage_references_config *const config)
```

Fill a configuration structure with the default configuration:

- 1.0V voltage reference typical value
- On demand control disabled
- The voltage reference and the temperature sensor are halted during standby sleep mode

Table 21-10. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_voltage_reference_set_config()`

Configure voltage reference.

```
void system_voltage_reference_set_config(  
    struct system_voltage_references_config *const config)
```

Configures voltage reference with the given configuration.

Table 21-11. Parameters

Data direction	Parameter name	Description
[in]	config	Voltage reference configuration structure containing the new config

Function `system_voltage_reference_enable()`

Enable the selected voltage reference.

```
void system_voltage_reference_enable(  
    const enum system_voltage_reference vref)
```

Enables the selected voltage reference source, making the voltage reference available on a pin as well as an input source to the analog peripherals.

Table 21-12. Parameters

Data direction	Parameter name	Description
[in]	vref	Voltage reference to enable

Function `system_voltage_reference_disable()`

Disable the selected voltage reference.

```
void system_voltage_reference_disable(  
    const enum system_voltage_reference vref)
```

Disables the selected voltage reference source.

Table 21-13. Parameters

Data direction	Parameter name	Description
[in]	vref	Voltage reference to disable

21.6.2.3. Battery Backup Power Switch

Function `system_battery_backup_power_switch_get_config_defaults()`

Retrieve the default configuration for battery backup power switch control.

```
void system_battery_backup_power_switch_get_config_defaults(  
    struct system_battery_backup_power_switch_config *const config)
```

Fills a configuration structure with the default configuration:

- The main Power Supply OK status is not available on the PSOK pin
- The device is not woken up when switched from battery backup power to main power
- The backup domain is always supplied by main power

Table 21-14. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_battery_backup_power_switch_set_config()`

Configure battery backup power switch.

```
void system_battery_backup_power_switch_set_config(  
    struct system_battery_backup_power_switch_config *const config)
```

Configures battery backup power switch with the given configuration.

Table 21-15. Parameters

Data direction	Parameter name	Description
[in]	config	Battery backup power switch configuration structure containing the new config

21.6.2.4. Output Pins in Backup Mode

Function `system_backup_pin_output_enable()`

Enable the backup pin output.

```
void system_backup_pin_output_enable(  
    enum system_backup_pin pin)
```

The output is enabled and driven by the SUPC.

Table 21-16. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Function `system_backup_pin_output_disable()`

Disable the backup pin output.

```
void system_backup_pin_output_disable(
    enum system_backup_pin pin)
```

The output is not enabled.

Table 21-17. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Function `system_backup_pin_output_is_enabled()`

Check if backup pin output is enabled.

```
bool system_backup_pin_output_is_enabled(
    enum system_backup_pin pin)
```

Table 21-18. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Returns

The enabled status.

Table 21-19. Return Values

Return value	Description
true	The output is enabled
false	The output is not enabled

Function `system_backup_pin_output_enable_rtc_toggle()`

Enable the backup pin toggle on RTC event.

```
void system_backup_pin_output_enable_rtc_toggle(
    enum system_backup_pin pin)
```

Toggle output on RTC event is enabled.

Table 21-20. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Function `system_backup_pin_output_disable_rtc_toggle()`

Disable the backup pin toggle on RTC event.

```
void system_backup_pin_output_disable_rtc_toggle(  
    enum system_backup_pin pin)
```

Toggle output on RTC event is disabled.

Table 21-21. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Function `system_backup_pin_output_set()`

Set the backup pin.

```
void system_backup_pin_output_set(  
    enum system_backup_pin pin)
```

Set the corresponding output pin.

Table 21-22. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Function `system_backup_pin_output_clear()`

Clear the backup pin.

```
void system_backup_pin_output_clear(  
    enum system_backup_pin pin)
```

Clear the corresponding output.

Table 21-23. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Function `system_backup_pin_output_get()`

Get the backup I/O input values.

```
bool system_backup_pin_output_get(  
    enum system_backup_pin pin)
```

Get the backup I/O data input values. If the corresponding pin is enabled, the I/O input value is given on the pin.

Table 21-24. Parameters

Data direction	Parameter name	Description
[in]	pin	Backup pin index

Returns

The backup I/O input level value.

21.6.2.5. Device Sleep Control

Function `system_set_sleepmode()`

Set the sleep mode of the device.

```
void system_set_sleepmode(  
    const enum system_sleepmode sleep_mode)
```

Sets the sleep mode of the device; the configured sleep mode will be entered upon the next call of the `system_sleep()` function.

For an overview of which systems are disabled in sleep for the different sleep modes, see [Sleep Modes](#).

Table 21-25. Parameters

Data direction	Parameter name	Description
[in]	sleep_mode	Sleep mode to configure for the next sleep operation

Function `system_sleep()`

Put the system to sleep waiting for interrupt.

```
void system_sleep( void )
```

Executes a device DSB (Data Synchronization Barrier) instruction to ensure all ongoing memory accesses have completed. Further, a WFI (Wait For Interrupt) instruction is executed to place the device into the sleep mode specified by `system_set_sleepmode`.

21.6.2.6. Performance Level Control

Function `system_switch_performance_level()`

Switch performance level.

```
enum status_code system_switch_performance_level(  
    const enum system_performance_level performance_level)
```

The bus frequency must be reduced prior to scaling down the performance level, in order to not exceed the maximum frequency allowed for the performance level.

When scaling up the performance level (for example from PL0 to PL2), the bus frequency can be increased first when the performance level transition is completed. Check the performance level status before increasing the frequency.

Table 21-26. Parameters

Data direction	Parameter name	Description
[in]	performance_level	Performance level to switch

Table 21-27. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid parameter
STATUS_OK	Successfully

Function `system_get_performance_level()`

Get performance level.

```
enum system_performance_level system_get_performance_level( void )
```

Get performance level.

Returns

Current performance level.

Function `system_get_performance_level_status()`

Get performance level status.

```
uint8_t system_get_performance_level_status( void )
```

Returns

Performance level status: Written to one when the performance level is ready.

Function `system_clear_performance_level_status()`

Clear performance level status.

```
void system_clear_performance_level_status( void )
```

Clear performance level status.

21.6.2.7. Standby Configuration

Function `system_standby_get_config_defaults()`

Retrieve the default configuration for standby.

```
void system_standby_get_config_defaults(
    struct system_standby_config *const config)
```

Fills a configuration structure with the default configuration for standby:

- Retention back biasing mode for HMCramCLP
- Retention back biasing mode for HMCramCHS
- Power domains PD0/PD1/PD2 are not linked
- Automatic VREG switching is used
- Dynamic power gating for power domain 1 is disabled
- Dynamic power gating for power domain 0 is disabled
- All power domains switching are handled by hardware

Table 21-28. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function system_standby_set_config()

Configure standby mode.

```
void system_standby_set_config(
    struct system_standby_config *const config)
```

Configures standby with the given configuration.

Table 21-29. Parameters

Data direction	Parameter name	Description
[in]	config	Standby configuration structure containing the new config

21.6.2.8. I/O Retention**Function system_io_retension_enable()**

Enable I/O retention.

```
void system_io_retension_enable( void )
```

Enable I/O retention. After waking up from Backup mode, I/O lines are held until the bit is written to 0.

Function system_io_retension_disable()

Disable I/O retention.

```
void system_io_retension_disable( void )
```

Disable IO retention. After waking up from Backup mode, I/O lines are not held.

21.6.2.9. Reset Control**Function system_reset()**

Reset the MCU.

```
void system_reset( void )
```

Resets the MCU and all associated peripherals and registers, except RTC, OSC32KCTRL, RSTC, GCLK (if WRTLOCK is set), and I/O retention state of PM.

Function system_get_reset_cause()

Get the reset cause.

```
enum system_reset_cause system_get_reset_cause( void )
```

Retrieves the cause of the last system reset.

Returns

An enum value indicating the cause of the last system reset.

21.6.2.10. Backup Exit Control

Function `system_get_backup_exit_source()`

Get the backup exit source.

```
enum system_reset_backup_exit_source system_get_backup_exit_source( void )
```

Get the backup exit source when a backup reset occurs.

Returns

An enum value indicating the latest backup exit source.

21.6.2.11. System Debugger

Function `system_is_debugger_present()`

Check if debugger is present.

```
bool system_is_debugger_present( void )
```

Check if debugger is connected to the onboard debug system (DAP).

Returns

A bool identifying if a debugger is present.

Table 21-30. Return Values

Return value	Description
true	Debugger is connected to the system
false	Debugger is not connected to the system

21.6.2.12. System Identification

Function `system_get_device_id()`

Retrieve the device identification signature.

```
uint32_t system_get_device_id( void )
```

Retrieves the signature of the current device.

Returns

Device ID signature as a 32-bit integer.

21.6.2.13. System Initialization

Function `system_init()`

Initialize system.

```
void system_init( void )
```

This function will call the various initialization functions within the system namespace. If a given optional system module is not available, the associated call will effectively be a NOP (No Operation).

Currently the following initialization functions are supported:

- System clock initialization (via the SYSTEM CLOCK sub-module)
- Board hardware initialization (via the Board module)
- Event system driver initialization (via the EVSYS module)
- External Interrupt driver initialization (via the EXTINT module)

21.6.3. Enumeration Definitions

21.6.3.1. Enum `system_backup_pin`

List of Backup input and output pins. If enabled ([system_backup_pin_output_enable](#)), the pins can be driven by the SUPC.

Table 21-31. Members

Enum value	Description
SYSTEM_BACKUP_PIN_PSOK	Power Supply OK status pin
SYSTEM_BACKUP_PIN_OUT_0	Backup output pin 0
SYSTEM_BACKUP_PIN_OUT_1	Backup output pin 1

21.6.3.2. Enum `system_battery_power_switch`

Enum for Battery power switch modes.

Table 21-32. Members

Enum value	Description
SYSTEM_BATTERY_POWER_SWITCH_NONE	The backup domain is always supplied by main power
SYSTEM_BATTERY_POWER_SWITCH_AUTOMATIC	The power switch is handled by the automatic power switch
SYSTEM_BATTERY_POWER_SWITCH_FORCED	The backup domain is always supplied by battery backup power
SYSTEM_BATTERY_POWER_SWITCH_BOD33	The power switch is handled by the BOD33

21.6.3.3. Enum `system_performance_level`

List of performance levels. Performance level technique consists of adjusting the regulator output voltage to reduce power consumption.

Table 21-33. Members

Enum value	Description
SYSTEM_PERFORMANCE_LEVEL_0	Performance level 0
SYSTEM_PERFORMANCE_LEVEL_2	Performance level 2

21.6.3.4. Enum system_ram_back_bias_mode

List of RAM back bias modes. By default, in standby sleep mode, RAM is in low power mode (back biased) if its power domain is in retention state. This behavior can be changed by configuring the Back Bias bit groups in STDBYCFG(STDBYCFG.BBIASxx).

Table 21-34. Members

Enum value	Description
SYSTEM_RAM_BACK_BIAS_RETENTION	Retention Back biasing mode
SYSTEM_RAM_BACK_BIAS_STANDBY	Standby Back Biasing mode
SYSTEM_RAM_BACK_BIAS_STANDBY_OFF	Standby OFF mode
SYSTEM_RAM_BACK_BIAS_OFF	Always OFF mode

21.6.3.5. Enum system_reset_backup_exit_source

List of possible backup exit source.

Table 21-35. Members

Enum value	Description
SYSTEM_RESET_BACKKUP_EXIT_RTC	The backup exit source was RTC interrupt.
SYSTEM_RESET_BACKKUP_EXIT_BBPS	The backup exit source was battery backup power switch

21.6.3.6. Enum system_reset_cause

List of possible reset causes of the system.

Table 21-36. Members

Enum value	Description
SYSTEM_RESET_CAUSE_BACKUP	The system was last reset by a backup reset
SYSTEM_RESET_CAUSE_SOFTWARE	The system was last reset by a software reset
SYSTEM_RESET_CAUSE_WDT	The system was last reset by the watchdog timer
SYSTEM_RESET_CAUSE_EXTERNAL_RESET	The system was last reset because the external reset line was pulled low
SYSTEM_RESET_CAUSE_BOD33	The system was last reset by the BOD VDD.
SYSTEM_RESET_CAUSE_BOD12	The system was last reset by the BOD CORE.
SYSTEM_RESET_CAUSE_POR	The system was last reset by the POR (Power on reset).

21.6.3.7. Enum system_sleepmode

List of available sleep modes in the device. A table of clocks available in different sleep modes can be found in [Sleep Modes](#).

Table 21-37. Members

Enum value	Description
SYSTEM_SLEEPMODE_IDLE	IDLE sleep mode
SYSTEM_SLEEPMODE_STANDBY	STANDBY sleep mode
SYSTEM_SLEEPMODE_BACKUP	BACKUP sleep mode
SYSTEM_SLEEPMODE_OFF	OFF sleep mode

21.6.3.8. Enum `system_voltage_reference`

List of available voltage references (VREF) that may be used within the device.

Table 21-38. Members

Enum value	Description
SYSTEM_VOLTAGE_REFERENCE_TEMPSENSE	Temperature sensor voltage reference
SYSTEM_VOLTAGE_REFERENCE_OUTPUT	Voltage reference output

21.6.3.9. Enum `system_voltage_references_sel`

Voltage references selection.

Table 21-39. Members

Enum value	Description
SYSTEM_VOLTAGE_REFERENCE_1V0	1.0V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_1V1	1.1V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_1V2	1.2V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_1V25	1.25V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_2V0	2.0V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_2V2	2.2V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_2V4	2.4V voltage reference typical value
SYSTEM_VOLTAGE_REFERENCE_2V5	2.5V voltage reference typical value

21.6.3.10. Enum `system_voltage_regulator_low_power_efficiency`

Low power mode efficiency.

Table 21-40. Members

Enum value	Description
SYSTEM_VOLTAGE_REGULATOR_LOW_POWER_EFFICIENCY_DEFAULT	The voltage regulator in Low power mode has the default efficiency and support the whole VDD range (1.62V to 3.6V)
SYSTEM_VOLTAGE_REGULATOR_LOW_POWER_EFFICIENCY_HIGHEST	The voltage regulator in Low power mode has the highest efficiency and support the limited VDD range (2.5V to 3.6V)

21.6.3.11. Enum `system_voltage_regulator_sel`

Voltage regulators selection. In active mode, the voltage regulator can be chosen on the fly between a LDO or a Buck converter.

Table 21-41. Members

Enum value	Description
SYSTEM_VOLTAGE_REGULATOR_LDO	The voltage regulator in active mode is a LDO voltage regulator
SYSTEM_VOLTAGE_REGULATOR_BUCK	The voltage regulator in active mode is a buck converter

21.7. Examples for SYSTEM Driver

This is a list of the available Quick Start Guides (QSGs) and example applications for [SAM System \(SYSTEM\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for Power Driver](#)

21.7.1. Quick Start Guide for Power Driver

List of supported boards:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM L22 Xplained Pro B

This example demonstrates how to use the power driver. BUTTON0 is used to wake up the system from the standby sleep mode and as an external wakeup pin to wake up the system from the backup sleep mode. The wakeup pin level is low. The I/O pins PB22/PB23 are used as GCLK0/GCLK1 outputs so that an oscilloscope can be used to monitor the clock frequencies.

After power-on-reset (POR), GCLK0 and GCLK1 runs at 4MHz and LED0 is turned on. After one second, LED0 is turned off and the system enters standby sleep mode. BUTTON0 can then be used to wake up

the system. After the system wakeup, LED0 is turned on, the performance level is switched to PL2, and the GCLK0 is increased to 48MHz. Further LED0 toggles two times and is turned off before the system enters BACKUP.

When BUTTON0 pushes, it connects to low level, system wakes up from the backup sleep mode, LED0 toggles four times. GCLK0/GCLK1 are running at 4MHz.

21.7.1.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Copy-paste the following setup code to your user application:

```
static void performance_level_switch_test(void)
{
    struct system_gclk_gen_config gclk_conf;

    /* Switch to PL2 */
    system_switch_performance_level(SYSTEM_PERFORMANCE_LEVEL_2);
    system_flash_set_waitstates(2);

    /* Switch GCLK0 to 48MHz */
    system_gclk_gen_get_config_defaults(&gclk_conf);
    gclk_conf.source_clock = SYSTEM_CLOCK_SOURCE_DFL;
    gclk_conf.division_factor = 1;
    gclk_conf.run_in_standby = false;
    gclk_conf.output_enable = true;
    system_gclk_gen_set_config(GCLK_GENERATOR_0, &gclk_conf);
}

static void config_clock_output_and_extwake_pin(void)
{
    struct system_pinmux_config pin_conf;
    system_pinmux_get_config_defaults(&pin_conf);

    pin_conf.mux_position = CONF_GCLK0_OUTPUT_PINMUX;
    pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_OUTPUT;
    system_pinmux_pin_set_config(CONF_GCLK0_OUTPUT_PIN, &pin_conf);
    pin_conf.mux_position = CONF_GCLK1_OUTPUT_PINMUX;
    system_pinmux_pin_set_config(CONF_GCLK1_OUTPUT_PIN, &pin_conf);
#ifdef SAML21 || SAMR30
    pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
    pin_conf.input_pull = SYSTEM_PINMUX_PIN_PULL_UP;
    pin_conf.mux_position = CONF_EXT_WAKEUP_PINMUX;
    system_pinmux_pin_set_config(CONF_EXT_WAKEUP_PIN, &pin_conf);
#endif
}

static void configure_extint_channel(void)
{
    struct extint_chan_conf config_extint_chan;
    extint_chan_get_config_defaults(&config_extint_chan);
    config_extint_chan.gpio_pin = BUTTON_0_EIC_PIN;
    config_extint_chan.gpio_pin_mux = BUTTON_0_EIC_MUX;
    config_extint_chan.gpio_pin_pull = EXTINT_PULL_UP;
    config_extint_chan.detection_criteria = EXTINT_DETECT_BOTH;
    extint_chan_set_config(BUTTON_0_EIC_LINE, &config_extint_chan);
}
```

```

extint_chan_enable_callback(BUTTON_0_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
system_interrupt_enable_global();
while (extint_chan_is_detected(BUTTON_0_EIC_LINE)) {
    extint_chan_clear_detected(BUTTON_0_EIC_LINE);
}
}

static void led_toggle_indication(uint32_t count)
{
    for (uint32_t i = 0; i < count; i++) {
        port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
        delay_ms(200);
        port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
        delay_ms(200);
    }
}

```

Workflow

1. Switch performance level to PL2.

```

static void performance_level_switch_test(void)
{
    struct system_gclk_gen_config gclk_conf;

    /* Switch to PL2 */
    system_switch_performance_level(SYSTEM_PERFORMANCE_LEVEL_2);
    system_flash_set_waitstates(2);

    /* Switch GCLK0 to 48MHz */
    system_gclk_gen_get_config_defaults(&gclk_conf);
    gclk_conf.source_clock = SYSTEM_CLOCK_SOURCE_DFLL;
    gclk_conf.division_factor = 1;
    gclk_conf.run_in_standby = false;
    gclk_conf.output_enable = true;
    system_gclk_gen_set_config(GCLK_GENERATOR_0, &gclk_conf);
}

```

2. Configure GCLK0/GCLK1 output pin and extwake pin.

```

static void config_clock_output_and_extwake_pin(void)
{
    struct system_pinmux_config pin_conf;
    system_pinmux_get_config_defaults(&pin_conf);

    pin_conf.mux_position = CONF_GCLK0_OUTPUT_PINMUX;
    pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_OUTPUT;
    system_pinmux_pin_set_config(CONF_GCLK0_OUTPUT_PIN, &pin_conf);
    pin_conf.mux_position = CONF_GCLK1_OUTPUT_PINMUX;
    system_pinmux_pin_set_config(CONF_GCLK1_OUTPUT_PIN, &pin_conf);
#ifdef SAML21 || SAMR30
    pin_conf.direction = SYSTEM_PINMUX_PIN_DIR_INPUT;
    pin_conf.input_pull = SYSTEM_PINMUX_PIN_PULL_UP;
    pin_conf.mux_position = CONF_EXT_WAKEUP_PINMUX;
    system_pinmux_pin_set_config(CONF_EXT_WAKEUP_PIN, &pin_conf);
#endif
}

```

3. Config external interrupt.

```

static void configure_extint_channel(void)
{

```

```

    struct extint_chan_conf config_extint_chan;
    extint_chan_get_config_defaults(&config_extint_chan);
    config_extint_chan.gpio_pin      = BUTTON_0_EIC_PIN;
    config_extint_chan.gpio_pin_mux  = BUTTON_0_EIC_MUX;
    config_extint_chan.gpio_pin_pull = EXTINT_PULL_UP;
    config_extint_chan.detection_criteria = EXTINT_DETECT_BOTH;
    extint_chan_set_config(BUTTON_0_EIC_LINE, &config_extint_chan);

    extint_chan_enable_callback(BUTTON_0_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
    system_interrupt_enable_global();
    while (extint_chan_is_detected(BUTTON_0_EIC_LINE)) {
        extint_chan_clear_detected(BUTTON_0_EIC_LINE);
    }
}

```

21.7.1.2. Use Case

Code

Copy-paste the following code to your user application:

```

#if SAML21 || SAMR30
/* Check if the RESET is caused by external wakeup pin */
if (system_get_reset_cause() == SYSTEM_RESET_CAUSE_BACKUP
    && system_get_backup_exit_source() ==
SYSTEM_RESET_BACKKUP_EXIT_EXTWAKE
    && (system_get_pin_wakeup_cause() & (1 << CONF_EXT_WAKEUP_PIN))
) {
    system_init();
    delay_init();
    config_clock_output_and_extwake_pin();
    /* Now GCLK0/GCLK1 are running at 4MHz , using LED0 ON/OFF as an indication */
    led_toggle_indication(4);
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
    while(1);
}
#endif

/* Disable I/O retention*/
system_io_retension_disable();

system_init();
delay_init();

port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

/* Config GCLK0/GCLK1 output pin and extwake pin */
config_clock_output_and_extwake_pin();

/* Config external interrupt for wakeup system from standby mode*/
configure_extint_channel();
delay_s(1);

/* Turn off LED0 before enter standby mode */
port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);

/* Set and enter standby mode, using default stanby config */
system_set_sleepmode(SYSTEM_SLEEPMODE_STANDBY);
system_sleep();

/* Detect BUTTON pressed and system wakeup from standby mode,

```

```

    turn on led */
port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

/* Disable external interrupt to avoid any impact */
extint_chan_disable_callback(BUTTON_0_EIC_LINE,
                             EXTINT_CALLBACK_TYPE_DETECT);
extint_chan_clear_detected(BUTTON_0_EIC_LINE);

/* Switch GCLK0 to 48MHz */
performance_level_switch_test();

/* GCLK0 is running at 48MHz and GCLK1 is running at 4MHz ,
   use led ON/OFF as an indication */
led_toggle_indication(2);

#if SAML21 || SAMR30
/* Set external wakeup pin polarity */
system_set_pin_wakeup_polarity_low(1<<CONF_EXT_WAKEUP_PIN);

/* Set external wakeup detector */
system_enable_pin_wakeup(1<<CONF_EXT_WAKEUP_PIN);
system_set_pin_wakeup_debounce_counter(SYSTEM_WAKEUP_DEBOUNCE_2CK32);

/* Enter BACKUP mode */
system_set_sleepmode(SYSTEM_SLEEPMODE_BACKUP);
system_sleep();

/* Now system is in BACKUP mode and wait for extwakeup pin to low */
#endif

```

Workflow

1. Check if the RESET is caused by external wakeup pin.

```

#if SAML21 || SAMR30
/* Check if the RESET is caused by external wakeup pin */
if (system_get_reset_cause() == SYSTEM_RESET_CAUSE_BACKUP
    && system_get_backup_exit_source() ==
SYSTEM_RESET_BACKUP_EXIT_EXTWAKE
    && (system_get_pin_wakeup_cause() & (1 << CONF_EXT_WAKEUP_PIN))
    ) {
    system_init();
    delay_init();
    config_clock_output_and_extwake_pin();
    /* Now GCLK0/GCLK1 are running at 4MHz , using LED0 ON/OFF as an indication */
    led_toggle_indication(4);
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
    while(1);
}
#endif

```

2. Check the standby mode and the backup sleep mode.

```

/* Disable I/O retention*/
system_io_retention_disable();

system_init();
delay_init();

port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

/* Config GCLK0/GCLK1 output pin and extwakeup pin */

```

```

config_clock_output_and_extwake_pin();

/* Config external interrupt for wakeup system from standby mode*/
configure_extint_channel();
delay_s(1);

/* Turn off LED0 before enter standby mode */
port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);

/* Set and enter standby mode, using default standby config */
system_set_sleepmode(SYSTEM_SLEEPMODE_STANDBY);
system_sleep();

/* Detect BUTTON pressed and system wakeup from standby mode,
   turn on led */
port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

/* Disable external interrupt to avoid any impact */
extint_chan_disable_callback(BUTTON_0_EIC_LINE,
                             EXTINT_CALLBACK_TYPE_DETECT);
extint_chan_clear_detected(BUTTON_0_EIC_LINE);

/* Switch GCLK0 to 48MHz */
performance_level_switch_test();

/* GCLK0 is running at 48MHz and GCLK1 is running at 4MHz ,
   use led ON/OFF as an indication */
led_toggle_indication(2);

#if SAML21 || SAMR30
/* Set external wakeup pin polarity */
system_set_pin_wakeup_polarity_low(1<<CONF_EXT_WAKEUP_PIN);

/* Set external wakeup detector */
system_enable_pin_wakeup(1<<CONF_EXT_WAKEUP_PIN);

system_set_pin_wakeup_debounce_counter(SYSTEM_WAKEUP_DEBOUNCE_2CK32);

/* Enter BACKUP mode */
system_set_sleepmode(SYSTEM_SLEEPMODE_BACKUP);
system_sleep();

/* Now system is in BACKUP mode and wait for extwakeup pin to low */
#endif

```

21.8. Extra Information for SYSTEM Driver

21.8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Definition
PM	Power Manager
SUPC	Supply Controller
RSTC	Reset Controller

21.8.2. Dependencies

This driver has the following dependencies:

- None

21.8.3. Errata

There are no errata related to this driver.

21.8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

22. SAM System Clock Management (SYSTEM CLOCK) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's clocking related functions. This includes the various clock sources, bus clocks, and generic clocks within the device, with functions to manage the enabling, disabling, source selection, and prescaling of clocks to various internal peripherals.

The following peripherals are used by this module:

- GCLK (Generic Clock Management)
- PM (Power Management)
- OSCCTRL (Oscillators Controller)
- OSC32KCTRL (32K Oscillators Controller)
- MCLK (Main Clock)

The following devices can use this module:

- Atmel | SMART SAM L22

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

22.1. Prerequisites

There are no prerequisites for this module.

22.2. Module Overview

The SAM devices contain a sophisticated clocking system, which is designed to give the maximum flexibility to the user application. This system allows a system designer to tune the performance and power consumption of the device in a dynamic manner, to achieve the best trade-off between the two for a particular application.

This driver provides a set of functions for the configuration and management of the various clock related functionalities within the device.

22.2.1. Clock Sources

The SAM devices have a number of master clock source modules, each of which being capable of producing a stabilized output frequency which can then be fed into the various peripherals and modules within the device.

Possible clock source modules include internal R/C oscillators, internal DFLL modules, as well as external crystal oscillators and/or clock inputs.

22.2.2. CPU / Bus Clocks

The CPU and AHB/APBx buses are clocked by the same physical clock source (referred in this module as the Main Clock). The CPU and bus clocks are divided into a number of clock domains. Each clock domain can run at different frequencies.

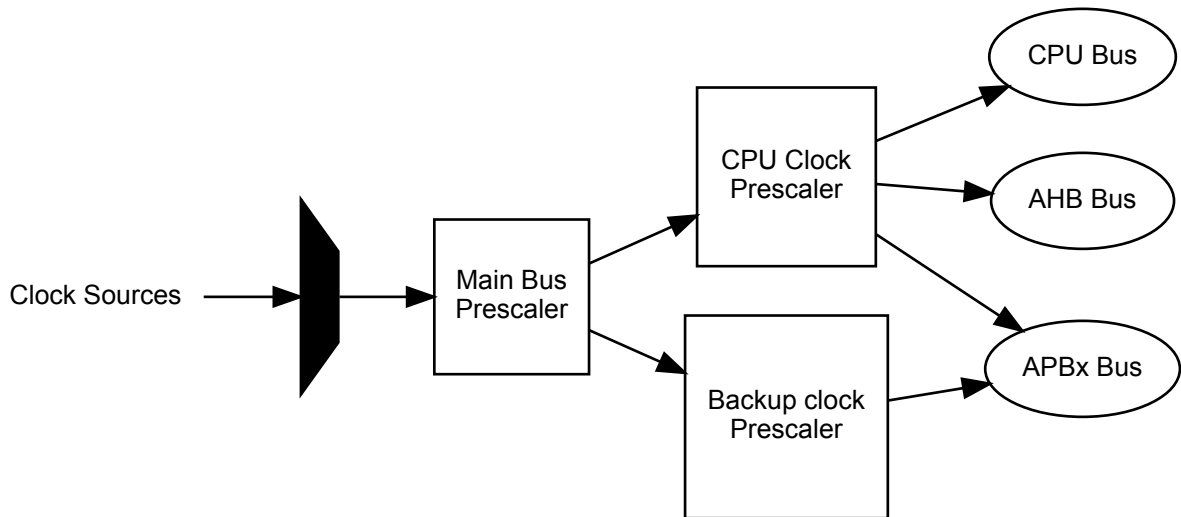
There are two clock domains:

- CPU Clock Domain
- BACKUP Clock Domain

Each clock domain (CPU, BUP) can be changed on the fly. To ensure correct operation, frequencies must be selected so that $BUPDIV \geq HSDIV$. Also, frequencies must never exceed the specified maximum frequency for each clock domain. A module may be connected to several clock domains (for instance, AHB and APB).

The general main clock tree for the CPU and associated buses is shown in [Figure 22-1](#).

Figure 22-1. CPU / Bus Clocks



22.2.3. Clock Masking

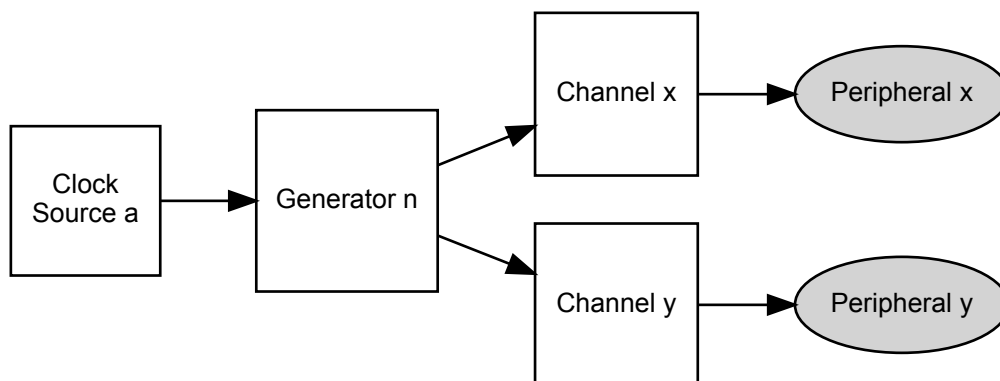
To save power, the input clock to one or more peripherals on the AHB and APBx buses can be masked away. When masked, no clock is passed into the module. Disabling of clocks of unused modules will prevent all access to the masked module, but will reduce the overall device power consumption.

22.2.4. Generic Clocks

Within the SAM devices are a number of Generic Clocks; these are used to provide clocks to the various peripheral clock domains in the device in a standardized manner. One or more master source clocks can be selected as the input clock to a Generic Clock Generator, which can prescale down the input frequency to a slower rate for use in a peripheral.

Additionally, a number of individually selectable Generic Clock Channels are provided, which multiplex and gate the various generator outputs for one or more peripherals within the device. This setup allows for a single common generator to feed one or more channels, which can then be enabled or disabled individually as required.

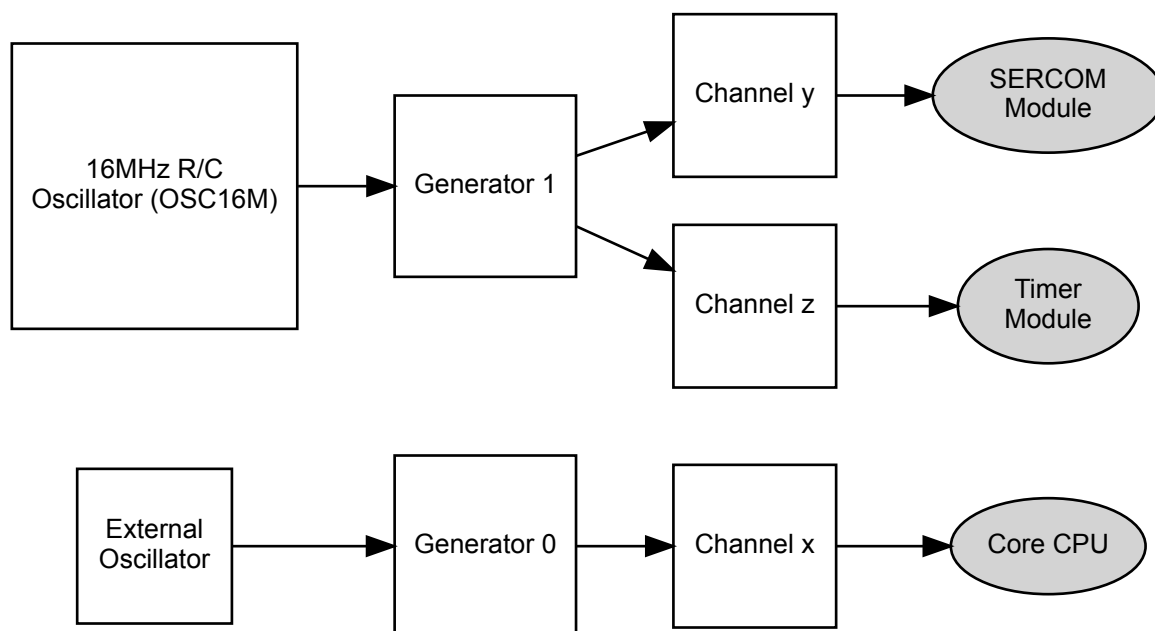
Figure 22-2. Generic Clocks



22.2.4.1. Clock Chain Example

An example setup of a complete clock chain within the device is shown in [Figure 22-3](#).

Figure 22-3. Clock Chain Example



22.2.4.2. Generic Clock Generators

Each Generic Clock generator within the device can source its input clock from one of the provided Source Clocks, and prescale the output for one or more Generic Clock Channels in a one-to-many relationship. The generators thus allow for several clocks to be generated of different frequencies, power usages, and accuracies, which can be turned on and off individually to disable the clocks to multiple peripherals as a group.

22.2.4.3. Generic Clock Channels

To connect a Generic Clock Generator to a peripheral within the device, a Generic Clock Channel is used. Each peripheral or peripheral group has an associated Generic Clock Channel, which serves as the clock input for the peripheral(s). To supply a clock to the peripheral module(s), the associated channel must be connected to a running Generic Clock Generator and the channel enabled.

22.3. Special Considerations

There are no special considerations for this module.

22.4. Extra Information

For extra information, see [Extra Information for SYSTEM CLOCK Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

22.5. Examples

For a list of examples related to this driver, see [Examples for System Clock Driver](#).

22.6. API Overview

22.6.1. Structure Definitions

22.6.1.1. Struct `system_clock_failure_detect`

Clock failure detect configuration structure.

Table 22-1. Members

Type	Name	Description
enum system_clock_failure_detect_div	<code>cfd_divider</code>	Clock failure detect safe clock prescaler
bool	<code>cfd_enable</code>	Clock failure detect enable
bool	<code>cfd_event_out</code>	Clock failure detect event output enable

22.6.1.2. Struct `system_clock_source_dfl_config`

DFLL oscillator configuration structure.

Table 22-2. Members

Type	Name	Description
enum system_clock_dfl_chill_cycle	<code>chill_cycle</code>	Enable chill cycle
uint8_t	<code>coarse_max_step</code>	Coarse adjustment maximum step size (Closed loop mode)
uint8_t	<code>coarse_value</code>	Coarse calibration value (Open loop mode)

Type	Name	Description
uint16_t	fine_max_step	Fine adjustment maximum step size (Closed loop mode)
uint16_t	fine_value	Fine calibration value (Open loop mode)
enum system_clock_dfl_loop_mode	loop_mode	Loop mode
uint16_t	multiply_factor	DFLL multiply factor (Closed loop mode)
bool	on_demand	Run On Demand. If this is set the DFLL won't run until requested by a peripheral
enum system_clock_dfl_quick_lock	quick_lock	Enable quick lock
bool	run_in_stanby	Run in stanby
enum system_clock_dfl_stable_tracking	stable_tracking	DFLL tracking after fine lock
enum system_clock_dfl_wakeup_lock	wakeup_lock	DFLL lock state on wakeup

22.6.1.3. Struct `system_clock_source_dppll_config`

DPLL oscillator configuration structure.

Table 22-3. Members

Type	Name	Description
enum system_clock_source_dppll_filter	filter	Filter type of the DPLL module
bool	lock_bypass	Bypass lock signal
enum system_clock_source_dppll_lock_time	lock_time	Lock time-out value of the DPLL module
bool	low_power_enable	Enable low power mode
bool	on_demand	Run On Demand. If this is set the DPLL won't run until requested by a peripheral
uint32_t	output_frequency	Output frequency of the clock
enum system_clock_source_dppll_prescaler	prescaler	DPLL prescaler
enum system_clock_source_dppll_reference_clock	reference_clock	Reference clock source of the DPLL module
uint16_t	reference_divider	Divider of reference clock
uint32_t	reference_frequency	Reference frequency of the clock

Type	Name	Description
bool	run_in_standby	Keep the DPLL enabled in standby sleep mode
bool	wake_up_fast	Wake up fast. If this is set DPLL output clock is enabled after the start-up time

22.6.1.4. Struct `system_clock_source_osc16m_config`

Internal 16MHz (nominal) oscillator configuration structure.

Table 22-4. Members

Type	Name	Description
enum system_osc16m_fsel	fsel	Internal 16MHz RC oscillator prescaler
bool	on_demand	Run On Demand. If this is set the OSC16M won't run until requested by a peripheral
bool	run_in_standby	Keep the OSC16M enabled in standby sleep mode

22.6.1.5. Struct `system_clock_source_osculp32k_config`

Internal 32KHz Ultra Low Power oscillator configuration structure.

Table 22-5. Members

Type	Name	Description
bool	enable_1khz_output	Enable 1KHz output
bool	enable_32khz_output	Enable 32KHz output
bool	write_once	Lock configuration after it has been written, a device reset will release the lock

22.6.1.6. Struct `system_clock_source_xosc32k_config`

External 32KHz oscillator clock configuration structure.

Table 22-6. Members

Type	Name	Description
struct system_clock_failure_detect	clock_failure_detect	Clock failure detect
bool	enable_1khz_output	Enable 1KHz output
bool	enable_32khz_output	Enable 32KHz output
enum system_clock_external	external_clock	External clock type
uint32_t	frequency	External clock/crystal frequency

Type	Name	Description
bool	on_demand	Run On Demand. If this is set the XOSC32K won't run until requested by a peripheral
bool	run_in_standby	Keep the XOSC32K enabled in standby sleep mode
enum system_xosc32k_startup	startup_time	Crystal oscillator start-up time
bool	write_once	Lock configuration after it has been written, a device reset will release the lock

22.6.1.7. Struct `system_clock_source_xosc_config`

External oscillator clock configuration structure.

Table 22-7. Members

Type	Name	Description
bool	auto_gain_control	Enable automatic amplitude gain control
struct system_clock_failure_detect	clock_failure_detect	Clock failure detect
enum system_clock_external	external_clock	External clock type
uint32_t	frequency	External clock/crystal frequency
bool	on_demand	Run On Demand. If this is set the XOSC won't run until requested by a peripheral
bool	run_in_standby	Keep the XOSC enabled in standby sleep mode
enum system_xosc_startup	startup_time	Crystal oscillator start-up time

22.6.1.8. Struct `system_gclk_chan_config`

Configuration structure for a Generic Clock channel. This structure should be initialized by the [system_gclk_chan_get_config_defaults\(\)](#) function before being modified by the user application.

Table 22-8. Members

Type	Name	Description
enum gclk_generator	source_generator	Generic Clock Generator source channel

22.6.1.9. Struct `system_gclk_gen_config`

Configuration structure for a Generic Clock Generator channel. This structure should be initialized by the [system_gclk_gen_get_config_defaults\(\)](#) function before being modified by the user application.

Table 22-9. Members

Type	Name	Description
uint32_t	division_factor	Integer division factor of the clock output compared to the input
bool	high_when_disabled	If <code>true</code> , the generator output level is high when disabled
bool	output_enable	If <code>true</code> , enables GCLK generator clock output to a GPIO pin
bool	run_in_standby	If <code>true</code> , the clock is kept enabled during device standby mode
uint8_t	source_clock	Source clock input channel index, see the system_clock_source

22.6.2. Function Definitions

22.6.2.1. External Oscillator Management

Function `system_clock_source_xosc_get_config_defaults()`

Retrieve the default configuration for XOSC.

```
void system_clock_source_xosc_get_config_defaults(
    struct system_clock_source_xosc_config *const config)
```

Fills a configuration structure with the default configuration for an external oscillator module:

- External Crystal
- Start-up time of 16384 external clock cycles
- Automatic crystal gain control mode enabled
- Frequency of 12MHz
- Don't run in STANDBY sleep mode
- Run only when requested by peripheral (on demand)
- Clock failure detect disabled
- CFD safe clock frequency is divided by 128
- Clock failure detect event output disabled

Table 22-10. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_clock_source_xosc_set_config()`

Configure the external oscillator clock source.

```
void system_clock_source_xosc_set_config(
    struct system_clock_source_xosc_config *const config)
```

Configures the external oscillator clock source with the given configuration settings.

Table 22-11. Parameters

Data direction	Parameter name	Description
[in]	config	External oscillator configuration structure containing the new config

Function `system_clock_source_xosc_set_switch_back()`

Controls XOSC output switch back to the external clock.

```
void system_clock_source_xosc_set_switch_back( void )
```

Controls XOSC output switch back to the external clock or crystal oscillator in case of clock recovery.

22.6.2.2. External 32KHz Oscillator Management

Function `system_clock_source_xosc32k_get_config_defaults()`

Retrieve the default configuration for XOSC32K.

```
void system_clock_source_xosc32k_get_config_defaults(  
    struct system_clock_source_xosc32k_config *const config)
```

Fills a configuration structure with the default configuration for an external 32KHz oscillator module:

- External Crystal
- Start-up time of 16384 external clock cycles
- Automatic crystal gain control mode disabled
- Frequency of 32.768KHz
- 1KHz clock output disabled
- 32KHz clock output enabled
- Don't run in STANDBY sleep mode
- Run only when requested by peripheral (on demand)
- Don't lock registers after configuration has been written
- Clock failure detect disabled
- CFD safe clock frequency is not divided
- Clock failure detect event output disabled

Table 22-12. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_clock_source_xosc32k_set_config()`

Configure the external 32KHz oscillator clock source.

```
void system_clock_source_xosc32k_set_config(  
    struct system_clock_source_xosc32k_config *const config)
```

Configures the external 32KHz oscillator clock source with the given configuration settings.

Table 22-13. Parameters

Data direction	Parameter name	Description
[in]	config	XOSC32K configuration structure containing the new config

Function `system_clock_source_xosc32k_set_switch_back()`

Controls XOSC32K output switch back to the external clock.

```
void system_clock_source_xosc32k_set_switch_back( void )
```


Controls XOSC32K output switch back to the external clock or OSCULP32K oscillator in case of clock recovery.

22.6.2.3. Internal Ultra Low Power 32KHz Oscillator Management

Function `system_clock_source_osculp32k_get_config_defaults()`

Retrieve the default configuration for OSCULP32K.

```
void system_clock_source_osculp32k_get_config_defaults(  
    struct system_clock_source_osculp32k_config *const config)
```

Fills a configuration structure with the default configuration for an internal Ultra Low Power 32KHz oscillator module:

- 1KHz clock output enabled
- 32KHz clock output enabled

Table 22-14. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_clock_source_osculp32k_set_config()`

Configure the internal OSCULP32K oscillator clock source.

```
void system_clock_source_osculp32k_set_config(  
    struct system_clock_source_osculp32k_config *const config)
```

Configures the Ultra Low Power 32KHz internal RC oscillator with the given configuration settings.

Table 22-15. Parameters

Data direction	Parameter name	Description
[in]	config	OSCULP32K configuration structure containing the new config

22.6.2.4. Internal 16MHz Oscillator Management

Function `system_clock_source_osc16m_get_config_defaults()`

Retrieve the default configuration for OSC16M.

```
void system_clock_source_osc16m_get_config_defaults(  
    struct system_clock_source_osc16m_config *const config)
```

Fills a configuration structure with the default configuration for an internal 16MHz (nominal) oscillator module:

- Clock output frequency select 4MHz
- Don't run in STANDBY sleep mode
- Run only when requested by peripheral (on demand)

Table 22-16. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_clock_source_osc16m_set_config()`

Configure the internal OSC16M oscillator clock source.

```
void system_clock_source_osc16m_set_config(  
    struct system_clock_source_osc16m_config *const config)
```

Configures the 16MHz (nominal) internal RC oscillator with the given configuration settings.

Note: Frequency selection can be done only when OSC16M is disabled.

Table 22-17. Parameters

Data direction	Parameter name	Description
[in]	config	OSC16M configuration structure containing the new config

22.6.2.5. Internal DFLL Management

Function `system_clock_source_dfll_get_config_defaults()`

Retrieve the default configuration for DFLL.

```
void system_clock_source_dfll_get_config_defaults(  
    struct system_clock_source_dfll_config *const config)
```

Fills a configuration structure with the default configuration for a DFLL oscillator module:

- Open loop mode
- QuickLock mode enabled
- Chill cycle enabled
- Output frequency lock maintained during device wake-up
- Continuous tracking of the output frequency
- Default tracking values at the mid-points for both coarse and fine tracking parameters
- Don't run in STANDBY sleep mode
- Run only when requested by peripheral (on demand)

Table 22-18. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_clock_source_dfll_set_config()`

Configure the DFLL clock source.

```
void system_clock_source_dfll_set_config(  
    struct system_clock_source_dfll_config *const config)
```

Configures the Digital Frequency Locked Loop clock source with the given configuration settings.

Note: The DFLL will be running when this function returns, as the DFLL module needs to be enabled in order to perform the module configuration.

Table 22-19. Parameters

Data direction	Parameter name	Description
[in]	config	DFLL configuration structure containing the new config

22.6.2.6. Clock Source Management

Function `system_clock_source_write_calibration()`

Writes the calibration values for a given oscillator clock source.

```
enum status_code system_clock_source_write_calibration(  
    const enum system_clock_source clock_source,  
    const uint16_t calibration_value,  
    const uint8_t freq_range)
```

Writes an oscillator calibration value to the given oscillator control registers. The acceptable ranges are:

For OSC32K:

- 7 bits (max. value 128) For OSC16MHZ:
- 8 bits (max. value 255) For OSCULP:
- 5 bits (max. value 32)

Note: The frequency range parameter applies only when configuring the 8MHz oscillator and will be ignored for the other oscillators.

Table 22-20. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to calibrate
[in]	calibration_value	Calibration value to write
[in]	freq_range	Frequency range (8MHz oscillator only)

Table 22-21. Return Values

Return value	Description
STATUS_OK	The calibration value was written successfully
STATUS_ERR_INVALID_ARG	The setting is not valid for selected clock source

Function `system_clock_source_enable()`

Enables a clock source.

```
enum status_code system_clock_source_enable(  
    const enum system_clock_source clock_source)
```

Enables a clock source which has been previously configured.

Table 22-22. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to enable

Table 22-23. Return Values

Return value	Description
STATUS_OK	Clock source was enabled successfully and is ready
STATUS_ERR_INVALID_ARG	The clock source is not available on this device

Function `system_clock_source_disable()`

Disables a clock source.

```
enum status_code system_clock_source_disable(
    const enum system_clock_source clk_source)
```

Disables a clock source that was previously enabled.

Table 22-24. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to disable

Table 22-25. Return Values

Return value	Description
STATUS_OK	Clock source was disabled successfully
STATUS_ERR_INVALID_ARG	An invalid or unavailable clock source was given

Function `system_clock_source_is_ready()`

Checks if a clock source is ready.

```
bool system_clock_source_is_ready(
    const enum system_clock_source clk_source)
```

Checks if a given clock source is ready to be used.

Table 22-26. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to check if ready

Returns

Ready state of the given clock source.

Table 22-27. Return Values

Return value	Description
true	Clock source is enabled and ready
false	Clock source is disabled or not yet ready

Function `system_clock_source_failure_is_detect()`

Checks if a clock failure is detected.

```
bool system_clock_source_failure_is_detect(  
    const enum system_clock_source clk_source)
```

Checks if a given clock failure is detected or not.

Table 22-28. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to check

Returns

Failure state of the given clock source.

Table 22-29. Return Values

Return value	Description
true	Clock source failure is detected
false	Clock source failure is not detected or function disabled

Function `system_clock_source_get_hz()`

Retrieve the frequency of a clock source.

```
uint32_t system_clock_source_get_hz(  
    const enum system_clock_source clk_source)
```

Determines the current operating frequency of a given clock source.

Table 22-30. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source

Returns

Frequency of the given clock source, in Hz.

22.6.2.7. Main Clock Management

Function `system_cpu_clock_set_divider()`

Set main CPU clock divider.

```
void system_cpu_clock_set_divider(  
    const enum system_main_clock_div divider)
```

Sets the clock divider used on the main clock to provide the CPU clock.

Table 22-31. Parameters

Data direction	Parameter name	Description
[in]	divider	CPU clock divider

Function `system_backup_clock_set_divider()`

Set Backup Clock divider.

```
void system_backup_clock_set_divider(  
    const enum system_main_clock_div divider)
```

Sets the clock divider used on the main clock to provide the CPU clock.

Table 22-32. Parameters

Data direction	Parameter name	Description
[in]	divider	CPU clock divider

Function `system_cpu_clock_get_hz()`

Retrieves the current frequency of the CPU core.

```
uint32_t system_cpu_clock_get_hz( void )
```

Retrieves the operating frequency of the CPU core, obtained from the main generic clock and the set CPU bus divider.

Returns

Current CPU frequency in Hz.

Function `system_backup_clock_get_hz()`

Retrieves the current frequency of backup clock.

```
uint32_t system_backup_clock_get_hz( void )
```

Retrieves the operating frequency of backup clock, obtained from backup clock and the set backup clock divider.

Returns

Current CPU frequency in Hz.

22.6.2.8. Bus Clock Masking

Function `system_ahb_clock_set_mask()`

Set bits in the clock mask for the AHB bus.

```
void system_ahb_clock_set_mask(  
    const uint32_t ahb_mask)
```

This function will set bits in the clock mask for the AHB bus. Any bits set to 1 will enable that clock, 0 bits in the mask will be ignored.

Table 22-33. Parameters

Data direction	Parameter name	Description
[in]	ahb_mask	AHB clock mask

Function `system_ahb_clock_clear_mask()`

Clear bits in the clock mask for the AHB bus.

```
void system_ahb_clock_clear_mask(  
    const uint32_t ahb_mask)
```

This function will clear bits in the clock mask for the AHB bus. Any bits set to 1 will disable that clock, zero bits in the mask will be ignored.

Table 22-34. Parameters

Data direction	Parameter name	Description
[in]	ahb_mask	AHB clock mask

Function `system_apb_clock_set_mask()`

Set bits in the clock mask for an APBx bus.

```
enum status_code system_apb_clock_set_mask(  
    const enum system_clock_apb_bus bus,  
    const uint32_t mask)
```

This function will set bits in the clock mask for an APBx bus. Any bits set to 1 will enable the corresponding module clock, zero bits in the mask will be ignored.

Table 22-35. Parameters

Data direction	Parameter name	Description
[in]	mask	APBx clock mask, a <code>SYSTEM_CLOCK_APB_APBx</code> constant from the device header files
[in]	bus	Bus to set clock mask bits for, a mask of <code>PM_APBxMASK_*</code> constants from the device header files

Returns

Status indicating the result of the clock mask change operation.

Table 22-36. Return Values

Return value	Description
<code>STATUS_ERR_INVALID_ARG</code>	Invalid bus given
<code>STATUS_OK</code>	The clock mask was set successfully

Function `system_apb_clock_clear_mask()`

Clear bits in the clock mask for an APBx bus.

```
enum status_code system_apb_clock_clear_mask(  
    const enum system_clock_apb_bus bus,  
    const uint32_t mask)
```

This function will clear bits in the clock mask for an APBx bus. Any bits set to 1 will disable the corresponding module clock, zero bits in the mask will be ignored.

Table 22-37. Parameters

Data direction	Parameter name	Description
[in]	mask	APBx clock mask, a <code>SYSTEM_CLOCK_APB_APBx</code> constant from the device header files
[in]	bus	Bus to clear clock mask bits

Returns

Status indicating the result of the clock mask change operation.

Table 22-38. Return Values

Return value	Description
<code>STATUS_ERR_INVALID_ARG</code>	Invalid bus ID was given
<code>STATUS_OK</code>	The clock mask was changed successfully

22.6.2.9. Internal DPLL Management**Function `system_clock_source_dpll_get_config_defaults()`**

Retrieve the default configuration for DPLL.

```
void system_clock_source_dpll_get_config_defaults(
    struct system_clock_source_dpll_config *const config)
```

Fills a configuration structure with the default configuration for a DPLL oscillator module:

- Run only when requested by peripheral (on demand)
- Don't run in STANDBY sleep mode
- Lock bypass disabled
- Fast wake up disabled
- Low power mode disabled
- Output frequency is 48MHz
- Reference clock frequency is 32768Hz
- Not divide reference clock
- Select REF0 as reference clock
- Set lock time to default mode
- Use default filter

Table 22-39. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to fill with default values

Function `system_clock_source_dpll_set_config()`

Configure the DPLL clock source.

```
void system_clock_source_dpll_set_config(
    struct system_clock_source_dpll_config *const config)
```

Configures the Digital Phase-Locked Loop clock source with the given configuration settings.

Note: The DPLL will be running when this function returns, as the DPLL module needs to be enabled in order to perform the module configuration.

Table 22-40. Parameters

Data direction	Parameter name	Description
[in]	config	DPLL configuration structure containing the new config

22.6.2.10. System Clock Initialization

Function `system_clock_init()`

Initialize clock system based on the configuration in `conf_clocks.h`.

```
void system_clock_init( void )
```

This function will apply the settings in `conf_clocks.h` when run from the user application. All clock sources and GCLK generators are running when this function returns.

Note: OSC16M is always enabled and if user selects other clocks for GCLK generators, the OSC16M default enable can be disabled after `system_clock_init`. Make sure the clock switch successfully before disabling OSC8M.

22.6.2.11. System Flash Wait States

Function `system_flash_set_waitstates()`

Set flash controller wait states.

```
void system_flash_set_waitstates(
    uint8_t wait_states)
```

Will set the number of wait states that are used by the onboard flash memory. The number of wait states depend on both device supply voltage and CPU speed. The required number of wait states can be found in the electrical characteristics of the device.

Table 22-41. Parameters

Data direction	Parameter name	Description
[in]	wait_states	Number of wait states to use for internal flash

22.6.2.12. Generic Clock Management

Function `system_gclk_init()`

Initializes the GCLK driver.

```
void system_gclk_init( void )
```

Initializes the Generic Clock module, disabling and resetting all active Generic Clock Generators and Channels to their power-on default values.

22.6.2.13. Generic Clock Management (Generators)

Function `system_gclk_gen_get_config_defaults()`

Initializes a Generic Clock Generator configuration structure to defaults.

```
void system_gclk_gen_get_config_defaults(
    struct system_gclk_gen_config *const config)
```

Initializes a given Generic Clock Generator configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is:

- The clock is generated undivided from the source frequency
- The clock generator output is low when the generator is disabled
- The input clock is sourced from input clock channel 0
- The clock will be disabled during sleep
- The clock output will not be routed to a physical GPIO pin

Table 22-42. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `system_gclk_gen_set_config()`

Writes a Generic Clock Generator configuration to the hardware module.

```
void system_gclk_gen_set_config(
    const uint8_t generator,
    struct system_gclk_gen_config *const config)
```

Writes out a given configuration of a Generic Clock Generator configuration to the hardware module.

Note: Changing the clock source on the fly (on a running generator) can take additional time if the clock source is configured to only run on-demand (ONDEMAND bit is set) and it is not currently running (no peripheral is requesting the clock source). In this case the GCLK will request the new clock while still keeping a request to the old clock source until the new clock source is ready.

Note: This function will not start a generator that is not already running; to start the generator, call [system_gclk_gen_enable\(\)](#) after configuring a generator.

Table 22-43. Parameters

Data direction	Parameter name	Description
[in]	generator	Generic Clock Generator index to configure
[in]	config	Configuration settings for the generator

Function `system_gclk_gen_enable()`

Enables a Generic Clock Generator that was previously configured.

```
void system_gclk_gen_enable(
    const uint8_t generator)
```

Starts the clock generation of a Generic Clock Generator that was previously configured via a call to [system_gclk_gen_set_config\(\)](#).

Table 22-44. Parameters

Data direction	Parameter name	Description
[in]	generator	Generic Clock Generator index to enable

Function `system_gclk_gen_disable()`

Disables a Generic Clock Generator that was previously enabled.

```
void system_gclk_gen_disable(  
    const uint8_t generator)
```

Stops the clock generation of a Generic Clock Generator that was previously started via a call to `system_gclk_gen_enable()`.

Table 22-45. Parameters

Data direction	Parameter name	Description
[in]	generator	Generic Clock Generator index to disable

Function `system_gclk_gen_is_enabled()`

Determines if the specified Generic Clock Generator is enabled.

```
bool system_gclk_gen_is_enabled(  
    const uint8_t generator)
```

Table 22-46. Parameters

Data direction	Parameter name	Description
[in]	generator	Generic Clock Generator index to check

Returns

The enabled status.

Table 22-47. Return Values

Return value	Description
true	The Generic Clock Generator is enabled
false	The Generic Clock Generator is disabled

22.6.2.14. Generic Clock Management (Channels)

Function `system_gclk_chan_get_config_defaults()`

Initializes a Generic Clock configuration structure to defaults.

```
void system_gclk_chan_get_config_defaults(  
    struct system_gclk_chan_config *const config)
```

Initializes a given Generic Clock configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- The clock is sourced from the Generic Clock Generator channel 0
- The clock configuration will not be write-locked when set

Table 22-48. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `system_gclk_chan_set_config()`

Writes a Generic Clock configuration to the hardware module.

```
void system_gclk_chan_set_config(
    const uint8_t channel,
    struct system_gclk_chan_config *const config)
```

Writes out a given configuration of a Generic Clock configuration to the hardware module. If the clock is currently running, it will be stopped.

Note: Once called the clock will not be running; to start the clock, call `system_gclk_chan_enable()` after configuring a clock channel.

Table 22-49. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock channel to configure
[in]	config	Configuration settings for the clock

Function `system_gclk_chan_enable()`

Enables a Generic Clock that was previously configured.

```
void system_gclk_chan_enable(
    const uint8_t channel)
```

Starts the clock generation of a Generic Clock that was previously configured via a call to `system_gclk_chan_set_config()`.

Table 22-50. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock channel to enable

Function `system_gclk_chan_disable()`

Disables a Generic Clock that was previously enabled.

```
void system_gclk_chan_disable(
    const uint8_t channel)
```

Stops the clock generation of a Generic Clock that was previously started via a call to `system_gclk_chan_enable()`.

Table 22-51. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock channel to disable

Function `system_gclk_chan_is_enabled()`

Determines if the specified Generic Clock channel is enabled.

```
bool system_gclk_chan_is_enabled(  
    const uint8_t channel)
```

Table 22-52. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock Channel index

Returns

The enabled status.

Table 22-53. Return Values

Return value	Description
true	The Generic Clock channel is enabled
false	The Generic Clock channel is disabled

Function `system_gclk_chan_lock()`

Locks a Generic Clock channel from further configuration writes.

```
void system_gclk_chan_lock(  
    const uint8_t channel)
```

Locks a generic clock channel from further configuration writes. It is only possible to unlock the channel configuration through a power on reset.

Table 22-54. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock channel to enable

Function `system_gclk_chan_is_locked()`

Determines if the specified Generic Clock channel is locked.

```
bool system_gclk_chan_is_locked(  
    const uint8_t channel)
```

Table 22-55. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock Channel index

Returns

The lock status.

Table 22-56. Return Values

Return value	Description
true	The Generic Clock channel is locked
false	The Generic Clock channel is not locked

22.6.2.15. Generic Clock Frequency Retrieval

Function `system_gclk_gen_get_hz()`

Retrieves the clock frequency of a Generic Clock generator.

```
uint32_t system_gclk_gen_get_hz(
    const uint8_t generator)
```

Determines the clock frequency (in Hz) of a specified Generic Clock generator, used as a source to a Generic Clock Channel module.

Table 22-57. Parameters

Data direction	Parameter name	Description
[in]	generator	Generic Clock Generator index

Returns

The frequency of the generic clock generator, in Hz.

Function `system_gclk_chan_get_hz()`

Retrieves the clock frequency of a Generic Clock channel.

```
uint32_t system_gclk_chan_get_hz(
    const uint8_t channel)
```

Determines the clock frequency (in Hz) of a specified Generic Clock channel, used as a source to a device peripheral module.

Table 22-58. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock Channel index

Returns

The frequency of the generic clock channel, in Hz.

22.6.3. Enumeration Definitions

22.6.3.1. Enum `gclk_generator`

List of Available GCLK generators. This enum is used in the peripheral device drivers to select the GCLK generator to be used for its operation.

The number of GCLK generators available is device dependent.

Table 22-59. Members

Enum value	Description
GCLK_GENERATOR_0	GCLK generator channel 0
GCLK_GENERATOR_1	GCLK generator channel 1
GCLK_GENERATOR_2	GCLK generator channel 2
GCLK_GENERATOR_3	GCLK generator channel 3
GCLK_GENERATOR_4	GCLK generator channel 4
GCLK_GENERATOR_5	GCLK generator channel 5
GCLK_GENERATOR_6	GCLK generator channel 6
GCLK_GENERATOR_7	GCLK generator channel 7
GCLK_GENERATOR_8	GCLK generator channel 8
GCLK_GENERATOR_9	GCLK generator channel 9
GCLK_GENERATOR_10	GCLK generator channel 10
GCLK_GENERATOR_11	GCLK generator channel 11
GCLK_GENERATOR_12	GCLK generator channel 12
GCLK_GENERATOR_13	GCLK generator channel 13
GCLK_GENERATOR_14	GCLK generator channel 14
GCLK_GENERATOR_15	GCLK generator channel 15
GCLK_GENERATOR_16	GCLK generator channel 16

22.6.3.2. Enum system_clock_apb_bus

Available bus clock domains on the APB bus.

Table 22-60. Members

Enum value	Description
SYSTEM_CLOCK_APB_APBA	Peripheral bus A on the APB bus
SYSTEM_CLOCK_APB_APBB	Peripheral bus B on the APB bus
SYSTEM_CLOCK_APB_APBC	Peripheral bus C on the APB bus

22.6.3.3. Enum system_clock_dfll_chill_cycle

DFLL chill cycle behavior modes of the DFLL module. A chill cycle is a period of time when the DFLL output frequency is not measured by the unit, to allow the output to stabilize after a change in the input clock source.

Table 22-61. Members

Enum value	Description
SYSTEM_CLOCK_DFLL_CHILL_CYCLE_ENABLE	Enable a chill cycle, where the DFLL output frequency is not measured
SYSTEM_CLOCK_DFLL_CHILL_CYCLE_DISABLE	Disable a chill cycle, where the DFLL output frequency is not measured

22.6.3.4. Enum system_clock_dfll_loop_mode

Available operating modes of the DFLL clock source module.

Table 22-62. Members

Enum value	Description
SYSTEM_CLOCK_DFLL_LOOP_MODE_OPEN	The DFLL is operating in open loop mode with no feedback
SYSTEM_CLOCK_DFLL_LOOP_MODE_CLOSED	The DFLL is operating in closed loop mode with frequency feedback from a low frequency reference clock

22.6.3.5. Enum system_clock_dfll_quick_lock

DFLL QuickLock settings for the DFLL module, to allow for a faster lock of the DFLL output frequency at the expense of accuracy.

Table 22-63. Members

Enum value	Description
SYSTEM_CLOCK_DFLL_QUICK_LOCK_ENABLE	Enable the QuickLock feature for looser lock requirements on the DFLL
SYSTEM_CLOCK_DFLL_QUICK_LOCK_DISABLE	Disable the QuickLock feature for strict lock requirements on the DFLL

22.6.3.6. Enum system_clock_dfll_stable_tracking

DFLL fine tracking behavior modes after a lock has been acquired.

Table 22-64. Members

Enum value	Description
SYSTEM_CLOCK_DFLL_STABLE_TRACKING_TRACK_AFTER_LOCK	Keep tracking after the DFLL has gotten a fine lock
SYSTEM_CLOCK_DFLL_STABLE_TRACKING_FIX_AFTER_LOCK	Stop tracking after the DFLL has gotten a fine lock

22.6.3.7. Enum system_clock_dfll_wakeup_lock

DFLL lock behavior modes on device wake-up from sleep.

Table 22-65. Members

Enum value	Description
SYSTEM_CLOCK_DFLL_WAKEUP_LOCK_KEEP	Keep DFLL lock when the device wakes from sleep
SYSTEM_CLOCK_DFLL_WAKEUP_LOCK_LOSE	Lose DFLL lock when the devices wakes from sleep

22.6.3.8. Enum `system_clock_external`

Available external clock source types.

Table 22-66. Members

Enum value	Description
SYSTEM_CLOCK_EXTERNAL_CRYSTAL	The external clock source is a crystal oscillator
SYSTEM_CLOCK_EXTERNAL_CLOCK	The connected clock source is an external logic level clock signal

22.6.3.9. Enum `system_clock_failure_detect_div`

The XOSC failure detect safe clock frequency is the OSC48M frequency divided by division ratios. The XOSC32K failure detect safe clock frequency is the OSCULP32K frequency divided by division ratios.

Note: The XOSC32K failure detect safe clock only can be divided by one and two.

Table 22-67. Members

Enum value	Description
SYSTEM_CFD_DIV_1	Divide safe clock frequency by 1
SYSTEM_CFD_DIV_2	Divide safe clock frequency by 2
SYSTEM_CFD_DIV_4	Divide safe clock frequency by 4
SYSTEM_CFD_DIV_8	Divide safe clock frequency by 8
SYSTEM_CFD_DIV_16	Divide safe clock frequency by 16
SYSTEM_CFD_DIV_32	Divide safe clock frequency by 32
SYSTEM_CFD_DIV_64	Divide safe clock frequency by 64
SYSTEM_CFD_DIV_128	Divide safe clock frequency by 128

22.6.3.10. Enum `system_clock_source`

Clock sources available to the GCLK generators.

Table 22-68. Members

Enum value	Description
SYSTEM_CLOCK_SOURCE_OSC16M	Internal 16MHz RC oscillator
SYSTEM_CLOCK_SOURCE_XOSC	External oscillator
SYSTEM_CLOCK_SOURCE_XOSC32K	External 32KHz oscillator
SYSTEM_CLOCK_SOURCE_DFLL	Digital Frequency Locked Loop (DFLL)
SYSTEM_CLOCK_SOURCE_ULP32K	Internal Ultra Low Power 32KHz oscillator
SYSTEM_CLOCK_SOURCE_GCLKIN	Generator input pad
SYSTEM_CLOCK_SOURCE_GCLKGEN1	Generic clock generator one output
SYSTEM_CLOCK_SOURCE_DPLL	Digital Phase Locked Loop (DPLL)

22.6.3.11. Enum system_clock_source_dpll_filter

Table 22-69. Members

Enum value	Description
SYSTEM_CLOCK_SOURCE_DPLL_FILTER_DEFAULT	Default filter mode
SYSTEM_CLOCK_SOURCE_DPLL_FILTER_LOW_BANDWIDTH_FILTER	Low bandwidth filter
SYSTEM_CLOCK_SOURCE_DPLL_FILTER_HIGH_BANDWIDTH_FILTER	High bandwidth filter
SYSTEM_CLOCK_SOURCE_DPLL_FILTER_HIGH_DAMPING_FILTER	High damping filter

22.6.3.12. Enum system_clock_source_dpll_lock_time

Table 22-70. Members

Enum value	Description
SYSTEM_CLOCK_SOURCE_DPLL_LOCK_TIME_DEFAULT	Set no time-out as default
SYSTEM_CLOCK_SOURCE_DPLL_LOCK_TIME_8MS	Set time-out if no lock within 8ms
SYSTEM_CLOCK_SOURCE_DPLL_LOCK_TIME_9MS	Set time-out if no lock within 9ms
SYSTEM_CLOCK_SOURCE_DPLL_LOCK_TIME_10MS	Set time-out if no lock within 10ms
SYSTEM_CLOCK_SOURCE_DPLL_LOCK_TIME_11MS	Set time-out if no lock within 11ms

22.6.3.13. Enum system_clock_source_dpll_prescaler

Table 22-71. Members

Enum value	Description
SYSTEM_CLOCK_SOURCE_DPLL_DIV_1	DPLL output is divided by 1
SYSTEM_CLOCK_SOURCE_DPLL_DIV_2	DPLL output is divided by 2
SYSTEM_CLOCK_SOURCE_DPLL_DIV_4	DPLL output is divided by 4

22.6.3.14. Enum system_clock_source_dpll_reference_clock

Table 22-72. Members

Enum value	Description
SYSTEM_CLOCK_SOURCE_DPLL_REFERENCE_CLOCK_XOSC32K	Select XOSC32K as clock reference
SYSTEM_CLOCK_SOURCE_DPLL_REFERENCE_CLOCK_XOSC	Select XOSC as clock reference
SYSTEM_CLOCK_SOURCE_DPLL_REFERENCE_CLOCK_GCLK	Select GCLK as clock reference

22.6.3.15. Enum system_main_clock_div

Available division ratios for the CPU and Lowpower and Backup clocks.

Table 22-73. Members

Enum value	Description
SYSTEM_MAIN_CLOCK_DIV_1	Divide Main clock by 1
SYSTEM_MAIN_CLOCK_DIV_2	Divide Main clock by 2
SYSTEM_MAIN_CLOCK_DIV_4	Divide Main clock by 4
SYSTEM_MAIN_CLOCK_DIV_8	Divide Main clock by 8
SYSTEM_MAIN_CLOCK_DIV_16	Divide Main clock by 16
SYSTEM_MAIN_CLOCK_DIV_32	Divide Main clock by 32
SYSTEM_MAIN_CLOCK_DIV_64	Divide Main clock by 64
SYSTEM_MAIN_CLOCK_DIV_128	Divide Main clock by 128

22.6.3.16. Enum system_osc16m_fsel

Available frequency selection for the internal 16MHz (nominal) system clock.

Table 22-74. Members

Enum value	Description
SYSTEM_OSC16M_4M	Frequency Selection 4MHz
SYSTEM_OSC16M_8M	Frequency Selection 8MHz
SYSTEM_OSC16M_12M	Frequency Selection 12MHz
SYSTEM_OSC16M_16M	Frequency Selection 16MHz

22.6.3.17. Enum system_xosc32k_startup

Available external 32KHz oscillator start-up times, as a number of external clock cycles.

Table 22-75. Members

Enum value	Description
SYSTEM_XOSC32K_STARTUP_2048	Wait 2048 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_4096	Wait 4096 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_16384	Wait 16384 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_32768	Wait 32768 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_65536	Wait 65536 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_131072	Wait 131072 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_262144	Wait 262144 clock cycles until the clock source is considered stable

22.6.3.18. Enum system_xosc_startup

Available external oscillator start-up times, as a number of external clock cycles.

Table 22-76. Members

Enum value	Description
SYSTEM_XOSC_STARTUP_1	Wait one clock cycle until the clock source is considered stable
SYSTEM_XOSC_STARTUP_2	Wait two clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_4	Wait four clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_8	Wait eight clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_16	Wait 16 clock cycles until the clock source is considered stable

Enum value	Description
SYSTEM_XOSC_STARTUP_32	Wait 32 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_64	Wait 64 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_128	Wait 128 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_256	Wait 256 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_512	Wait 512 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_1024	Wait 1024 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_2048	Wait 2048 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_4096	Wait 4096 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_8192	Wait 8192 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_16384	Wait 16384 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_32768	Wait 32768 clock cycles until the clock source is considered stable

22.7. Extra Information for SYSTEM CLOCK Driver

22.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
DFLL	Digital Frequency Locked Loop
MUX	Multiplexer
MCLK	Main Clock
OSC16M	Internal 16MHz Oscillator
PLL	Phase Locked Loop
OSC	Oscillator
XOSC	External Oscillator
XOSC32K	External 32KHz Oscillator
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
DPLL	Digital Phase Locked Loop

22.7.2. Dependencies

This driver has the following dependencies:

- None

22.7.3. Errata

- This driver implements experimental workaround for errata 9905
"The DFLL clock must be requested before being configured. Otherwise a write access to a DFLL register can freeze the device." This driver will enable and configure the DFLL before the ONDEMAND bit is set.

22.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

22.8. Examples for System Clock Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM System Clock Management \(SYSTEM CLOCK\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for SYSTEM CLOCK - Basic](#)
- [Quick Start Guide for SYSTEM CLOCK - GCLK Configuration](#)

22.8.1. Quick Start Guide for SYSTEM CLOCK - Basic

In this case we apply the following configuration:

- RC8MHz (internal 8MHz RC oscillator)
 - Divide by four, giving a frequency of 2MHz
- DFLL (Digital frequency locked loop)
 - Open loop mode
 - 48MHz frequency
- CPU clock
 - Use two wait states when reading from flash memory
 - Use the DFLL, configured to 48MHz

22.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your application:

```
void configure_extosc32k(void)
{
    struct system_clock_source_xosc32k_config config_ext32k;
    system_clock_source_xosc32k_get_config_defaults(&config_ext32k);

    config_ext32k.startup_time = SYSTEM_XOSC32K_STARTUP_4096;

    system_clock_source_xosc32k_set_config(&config_ext32k);
}

#if (!SAMC21)
void configure_dfll_open_loop(void)
{
    struct system_clock_source_dfll_config config_dfll;
    system_clock_source_dfll_get_config_defaults(&config_dfll);

    system_clock_source_dfll_set_config(&config_dfll);
}
#endif
```

Workflow

1. Create a EXTOSC32K module configuration struct, which can be filled out to adjust the configuration of the external 32KHz oscillator channel.

```
struct system_clock_source_xosc32k_config config_ext32k;
```

2. Initialize the oscillator configuration struct with the module's default values.

```
system_clock_source_xosc32k_get_config_defaults(&config_ext32k);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the EXTOSC32K module configuration struct to require a start-up time of 4096 clock cycles.

```
config_ext32k.startup_time = SYSTEM_XOSC32K_STARTUP_4096;
```

4. Write the new configuration to the EXTOSC32K module.

```
system_clock_source_xosc32k_set_config(&config_ext32k);
```

5. Create a DFLL module configuration struct, which can be filled out to adjust the configuration of the external 32KHz oscillator channel.

```
struct system_clock_source_dfll_config config_dfll;
```

6. Initialize the DFLL oscillator configuration struct with the module's default values.

```
system_clock_source_dfll_get_config_defaults(&config_dfll);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

7. Write the new configuration to the DFLL module.

```
system_clock_source_dfll_set_config(&config_dfll);
```

22.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
/* Configure the external 32KHz oscillator */
configure_extosc32k();

/* Enable the external 32KHz oscillator */
enum status_code osc32k_status =
    system_clock_source_enable(SYSTEM_CLOCK_SOURCE_XOSC32K);

if (osc32k_status != STATUS_OK) {
    /* Error enabling the clock source */
}

#if (!SAMC21)
    /* Configure the DFLL in open loop mode using default values */
    configure_dfll_open_loop();

/* Enable the DFLL oscillator */
enum status_code dfll_status =
    system_clock_source_enable(SYSTEM_CLOCK_SOURCE_DFLL);

if (dfll_status != STATUS_OK) {
    /* Error enabling the clock source */
}

/* Configure flash wait states before switching to high frequency clock */
system_flash_set_waitstates(2);

/* Change system clock to DFLL */
struct system_gclk_gen_config config_gclock_gen;
system_gclk_gen_get_config_defaults(&config_gclock_gen);
config_gclock_gen.source_clock = SYSTEM_CLOCK_SOURCE_DFLL;
config_gclock_gen.division_factor = 1;
system_gclk_gen_set_config(GCLK_GENERATOR_0, &config_gclock_gen);
#endif
```

Workflow

1. Configure the external 32KHz oscillator source using the previously defined setup function.

```
configure_extosc32k();
```

2. Enable the configured external 32KHz oscillator source.

```
enum status_code osc32k_status =
    system_clock_source_enable(SYSTEM_CLOCK_SOURCE_XOSC32K);

if (osc32k_status != STATUS_OK) {
    /* Error enabling the clock source */
}
```

3. Configure the DFLL oscillator source using the previously defined setup function.

```
configure_dfll_open_loop();
```

4. Enable the configured DFLL oscillator source.

```
enum status_code dfll_status =
    system_clock_source_enable(SYSTEM_CLOCK_SOURCE_DFLL);

if (dfll_status != STATUS_OK) {
```



```

    /* Error enabling the clock source */
}

```

5. Configure the flash wait states to have two wait states per read, as the high speed DFLL will be used as the system clock. If insufficient wait states are used, the device may crash randomly due to misread instructions.

```

system_flash_set_waitstates(2);

```

6. Switch the system clock source to the DFLL, by reconfiguring the main clock generator.

```

struct system_gclk_gen_config config_gclock_gen;
system_gclk_gen_get_config_defaults(&config_gclock_gen);
config_gclock_gen.source_clock = SYSTEM_CLOCK_SOURCE_DFLL;
config_gclock_gen.division_factor = 1;
system_gclk_gen_set_config(GCLK_GENERATOR_0, &config_gclock_gen);

```

22.8.2. Quick Start Guide for SYSTEM CLOCK - GCLK Configuration

In this use-case, the GCLK module is configured for:

- One generator attached to the internal 8MHz RC oscillator clock source
- Generator output equal to input frequency divided by a factor of 128
- One channel (connected to the TC0 module) enabled with the enabled generator selected

This use-case configures a clock channel to output a clock for a peripheral within the device, by first setting up a clock generator from a master clock source, and then linking the generator to the desired channel. This clock can then be used to clock a module within the device.

22.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```

void configure_gclock_generator(void)
{
    struct system_gclk_gen_config gclock_gen_conf;
    system_gclk_gen_get_config_defaults(&gclock_gen_conf);

    #if (SAML21) || (SAML22) || (SAMR30)
        gclock_gen_conf.source_clock = SYSTEM_CLOCK_SOURCE_OSC16M;
        gclock_gen_conf.division_factor = 128;
    #elif (SAMC21)
        gclock_gen_conf.source_clock = SYSTEM_CLOCK_SOURCE_OSC48M;
        gclock_gen_conf.division_factor = 128;
    #else
        gclock_gen_conf.source_clock = SYSTEM_CLOCK_SOURCE_OSC8M;
        gclock_gen_conf.division_factor = 128;
    #endif

    system_gclk_gen_set_config(GCLK_GENERATOR_1, &gclock_gen_conf);
    system_gclk_gen_enable(GCLK_GENERATOR_1);
}

void configure_gclock_channel(void)
{
    struct system_gclk_chan_config gclk_chan_conf;
    system_gclk_chan_get_config_defaults(&gclk_chan_conf);
}

```

```

    gclk_chan_conf.source_generator = GCLK_GENERATOR_1;
#if (SAMD10) || (SAMD11) || SAMR30
    system_gclk_chan_set_config(TC1_GCLK_ID, &gclk_chan_conf);

    system_gclk_chan_enable(TC1_GCLK_ID);
#else
    system_gclk_chan_set_config(TC3_GCLK_ID, &gclk_chan_conf);

    system_gclk_chan_enable(TC3_GCLK_ID);
#endif
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_gclock_generator();
configure_gclock_channel();

```

Workflow

1. Create a GCLK generator configuration struct, which can be filled out to adjust the configuration of a single clock generator.

```
struct system_gclk_gen_config gclock_gen_conf;
```

2. Initialize the generator configuration struct with the module's default values.

```
system_gclk_gen_get_config_defaults(&gclock_gen_conf);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to request the master clock source channel 0 is used as the source of the generator, and set the generator output prescaler to divide the input clock by a factor of 128.

```
gclock_gen_conf.source_clock = SYSTEM_CLOCK_SOURCE_OSC16M;
gclock_gen_conf.division_factor = 128;
```

4. Configure the generator using the configuration structure.

```
system_gclk_gen_set_config(GCLK_GENERATOR_1, &gclock_gen_conf);
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

5. Enable the generator once it has been properly configured, to begin clock generation.

```
system_gclk_gen_enable(GCLK_GENERATOR_1);
```

6. Create a GCLK channel configuration struct, which can be filled out to adjust the configuration of a single generic clock channel.

```
struct system_gclk_chan_config gclk_chan_conf;
```

7. Initialize the channel configuration struct with the module's default values.

```
system_gclk_chan_get_config_defaults(&gclk_chan_conf);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

8. Adjust the configuration struct to request the previously configured and enabled clock generator is used as the clock source for the channel.

```
gclk_chan_conf.source_generator = GCLK_GENERATOR_1;
```

9. Configure the channel using the configuration structure.

```
system_gclk_chan_set_config(TC1_GCLK_ID, &gclk_chan_conf);
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

10. Enable the channel once it has been properly configured, to output the clock to the channel's peripheral module consumers.

```
system_gclk_chan_enable(TC1_GCLK_ID);
```

22.8.2.2. Use-case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    /* Nothing to do */  
}
```

Workflow

1. As the clock is generated asynchronously to the system core, no special extra application code is required.

23. SAM System Interrupt (SYSTEM INTERRUPT) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of internal software and hardware interrupts/exceptions.

The following peripheral is used by this module:

- NVIC (Nested Vector Interrupt Controller)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

23.1. Prerequisites

There are no prerequisites for this module.

23.2. Module Overview

The ARM® Cortex® M0+ core contains an interrupt and exception vector table, which can be used to configure the device's interrupt handlers; individual interrupts and exceptions can be enabled and disabled, as well as configured with a variable priority.

This driver provides a set of wrappers around the core interrupt functions, to expose a simple API for the management of global and individual interrupts within the device.

23.2.1. Critical Sections

In some applications it is important to ensure that no interrupts may be executed by the system whilst a critical portion of code is being run; for example, a buffer may be copied from one context to another - during which interrupts must be disabled to avoid corruption of the source buffer contents until the copy has completed. This driver provides a basic API to enter and exit nested critical sections, so that global interrupts can be kept disabled for as long as necessary to complete a critical application code section.

23.2.2. Software Interrupts

For some applications, it may be desirable to raise a module or core interrupt via software. For this reason, a set of APIs to set an interrupt or exception as pending are provided to the user application.

23.3. Special Considerations

Interrupts from peripherals in the SAM devices are on a per-module basis; an interrupt raised from any source within a module will cause a single, module-common handler to execute. It is the user application or driver's responsibility to de-multiplex the module-common interrupt to determine the exact interrupt cause.

23.4. Extra Information

For extra information, see [Extra Information for SYSTEM INTERRUPT Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

23.5. Examples

For a list of examples related to this driver, see [Examples for SYSTEM INTERRUPT Driver](#).

23.6. API Overview

23.6.1. Function Definitions

23.6.1.1. Critical Section Management

Function `system_interrupt_enter_critical_section()`

Enters a critical section.

```
void system_interrupt_enter_critical_section( void )
```

Disables global interrupts. To support nested critical sections, an internal count of the critical section nesting will be kept, so that global interrupts are only re-enabled upon leaving the outermost nested critical section.

Function `system_interrupt_leave_critical_section()`

Leaves a critical section.

```
void system_interrupt_leave_critical_section( void )
```

Enables global interrupts. To support nested critical sections, an internal count of the critical section nesting will be kept, so that global interrupts are only re-enabled upon leaving the outermost nested critical section.

23.6.1.2. Interrupt Enabling/Disabling

Function `system_interrupt_is_global_enabled()`

Check if global interrupts are enabled.

```
bool system_interrupt_is_global_enabled( void )
```

Checks if global interrupts are currently enabled.

Returns

A boolean that identifies if the global interrupts are enabled or not.

Table 23-1. Return Values

Return value	Description
true	Global interrupts are currently enabled
false	Global interrupts are currently disabled

Function `system_interrupt_enable_global()`

Enables global interrupts.

```
void system_interrupt_enable_global( void )
```

Enables global interrupts in the device to fire any enabled interrupt handlers.

Function `system_interrupt_disable_global()`

Disables global interrupts.

```
void system_interrupt_disable_global( void )
```

Disabled global interrupts in the device, preventing any enabled interrupt handlers from executing.

Function `system_interrupt_is_enabled()`

Checks if an interrupt vector is enabled or not.

```
bool system_interrupt_is_enabled(  
    const enum system_interrupt_vector vector)
```

Checks if a specific interrupt vector is currently enabled.

Table 23-2. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector number to check

Returns

A variable identifying if the requested interrupt vector is enabled.

Table 23-3. Return Values

Return value	Description
true	Specified interrupt vector is currently enabled
false	Specified interrupt vector is currently disabled

Function `system_interrupt_enable()`

Enable interrupt vector.

```
void system_interrupt_enable(  
    const enum system_interrupt_vector vector)
```

Enables execution of the software handler for the requested interrupt vector.

Table 23-4. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector to enable

Function `system_interrupt_disable()`

Disable interrupt vector.

```
void system_interrupt_disable(  
    const enum system_interrupt_vector vector)
```

Disables execution of the software handler for the requested interrupt vector.

Table 23-5. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector to disable

23.6.1.3. Interrupt State Management

Function `system_interrupt_get_active()`

Get active interrupt (if any).

```
enum system_interrupt_vector system_interrupt_get_active( void )
```

Return the vector number for the current executing software handler, if any.

Returns

Interrupt number that is currently executing.

Function `system_interrupt_is_pending()`

Check if a interrupt line is pending.

```
bool system_interrupt_is_pending(  
    const enum system_interrupt_vector vector)
```

Checks if the requested interrupt vector is pending.

Table 23-6. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector number to check

Returns

A boolean identifying if the requested interrupt vector is pending.

Table 23-7. Return Values

Return value	Description
true	Specified interrupt vector is pending
false	Specified interrupt vector is not pending

Function `system_interrupt_set_pending()`

Set a interrupt vector as pending.

```
enum status_code system_interrupt_set_pending(
    const enum system_interrupt_vector vector)
```

Set the requested interrupt vector as pending (i.e. issues a software interrupt request for the specified vector). The software handler will be handled (if enabled) in a priority order based on vector number and configured priority settings.

Table 23-8. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector number which is set as pending

Returns

Status code identifying if the vector was successfully set as pending.

Table 23-9. Return Values

Return value	Description
STATUS_OK	If no error was detected
STATUS_INVALID_ARG	If an unsupported interrupt vector number was given

Function `system_interrupt_clear_pending()`

Clear pending interrupt vector.

```
enum status_code system_interrupt_clear_pending(
    const enum system_interrupt_vector vector)
```

Clear a pending interrupt vector, so the software handler is not executed.

Table 23-10. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector number to clear

Returns

A status code identifying if the interrupt pending state was successfully cleared.

Table 23-11. Return Values

Return value	Description
STATUS_OK	If no error was detected
STATUS_INVALID_ARG	If an unsupported interrupt vector number was given

23.6.1.4. Interrupt Priority Management

Function `system_interrupt_set_priority()`

Set interrupt vector priority level.

```
enum status_code system_interrupt_set_priority(
    const enum system_interrupt_vector vector,
    const enum system_interrupt_priority_level priority_level)
```

Set the priority level of an external interrupt or exception.

Table 23-12. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector to change
[in]	priority_level	New vector priority level to set

Returns

Status code indicating if the priority level of the interrupt was successfully set.

Table 23-13. Return Values

Return value	Description
STATUS_OK	If no error was detected
STATUS_INVALID_ARG	If an unsupported interrupt vector number was given

Function `system_interrupt_get_priority()`

Get interrupt vector priority level.

```
enum system_interrupt_priority_level system_interrupt_get_priority(
    const enum system_interrupt_vector vector)
```

Retrieves the priority level of the requested external interrupt or exception.

Table 23-14. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector of which the priority level will be read

Returns

Currently configured interrupt priority level of the given interrupt vector.

23.6.2. Enumeration Definitions

23.6.2.1. Enum `system_interrupt_priority_level`

Table of all possible interrupt and exception vector priorities within the device.

Table 23-15. Members

Enum value	Description
SYSTEM_INTERRUPT_PRIORITY_LEVEL_0	Priority level 0, the highest possible interrupt priority
SYSTEM_INTERRUPT_PRIORITY_LEVEL_1	Priority level 1
SYSTEM_INTERRUPT_PRIORITY_LEVEL_2	Priority level 2
SYSTEM_INTERRUPT_PRIORITY_LEVEL_3	Priority level 3, the lowest possible interrupt priority

23.6.2.2. Enum `system_interrupt_vector_samc21`

Table of all possible interrupt and exception vector indexes within the SAM C20/C21 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-16. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_SYSTEM	Interrupt vector index for MCLK, OSCCTRL, OSC32KCTRL, PAC, PM, SUPC, TAL peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_FREQM	Interrupt vector index for Frequency Meter peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCCn	Interrupt vector index for a Timer/Counter Control peripheral interrupt. Each specific device may contain several TCC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TCC0).
SYSTEM_INTERRUPT_MODULE_TCn	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).

23.6.2.3. Enum system_interrupt_vector_samd09

Table of all possible interrupt and exception vector indexes within the SAM D09 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-17. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_PM	Interrupt vector index for a Power Manager peripheral interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_SYSCTRL	Interrupt vector index for a System Control peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCn	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt

23.6.2.4. Enum system_interrupt_vector_samd1x

Table of all possible interrupt and exception vector indexes within the SAM D10/D11 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-18. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception

Enum value	Description
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_PM	Interrupt vector index for a Power Manager peripheral interrupt
SYSTEM_INTERRUPT_MODULE_SYSCTRL	Interrupt vector index for a System Control peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCCn	Interrupt vector index for a Timer/Counter Control peripheral interrupt. Each specific device may contain several TCC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TCC0).

Enum value	Description
SYSTEM_INTERRUPT_MODULE_TCN	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_DAC	Interrupt vector index for a Digital-to-Analog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_PTC	Interrupt vector index for a Peripheral Touch Controller peripheral interrupt

23.6.2.5. Enum system_interrupt_vector_samd20

Table of all possible interrupt and exception vector indexes within the SAM D20 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-19. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_PM	Interrupt vector index for a Power Manager peripheral interrupt
SYSTEM_INTERRUPT_MODULE_SYSCtrl	Interrupt vector index for a System Control peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCN	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC0).
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_DAC	Interrupt vector index for a Digital-to-Analog peripheral interrupt

23.6.2.6. Enum system_interrupt_vector_samd21

Table of all possible interrupt and exception vector indexes within the SAM D21 device. Check peripherals configuration in SAM D21 datasheet for available vector index for specific device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-20. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt

Enum value	Description
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_PM	Interrupt vector index for a Power Manager peripheral interrupt
SYSTEM_INTERRUPT_MODULE_SYSCTRL	Interrupt vector index for a System Control peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_USB	Interrupt vector index for a Universal Serial Bus interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCCn	Interrupt vector index for a Timer/Counter Control peripheral interrupt. Each specific device may contain several TCC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TCC0).

Enum value	Description
SYSTEM_INTERRUPT_MODULE_TCn	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt
SYSTEM_INTERRUPT_MODULE_DAC	Interrupt vector index for a Digital-to-Analog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_PTC	Interrupt vector index for a Peripheral Touch Controller peripheral interrupt
SYSTEM_INTERRUPT_MODULE_I2S	Interrupt vector index for a Inter-IC Sound Interface peripheral interrupt
SYSTEM_INTERRUPT_MODULE_AC1	Interrupt vector index for an Analog Comparator 1 peripheral interrupt

23.6.2.7. Enum system_interrupt_vector_samda1

Table of all possible interrupt and exception vector indexes within the SAM DA1 device. Check peripherals configuration in SAM DA1 datasheet for available vector index for specific device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-21. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_PM	Interrupt vector index for a Power Manager peripheral interrupt
SYSTEM_INTERRUPT_MODULE_SYSCTRL	Interrupt vector index for a System Control peripheral interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_USB	Interrupt vector index for a Universal Serial Bus interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCCn	Interrupt vector index for a Timer/Counter Control peripheral interrupt. Each specific device may contain several TCC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TCC0).
SYSTEM_INTERRUPT_MODULE_TCn	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_DAC	Interrupt vector index for a Digital-to-Analog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_PTC	Interrupt vector index for a Peripheral Touch Controller peripheral interrupt
SYSTEM_INTERRUPT_MODULE_I2S	Interrupt vector index for a Inter-IC Sound Interface peripheral interrupt
SYSTEM_INTERRUPT_MODULE_AC1	Interrupt vector index for an Analog Comparator 1 peripheral interrupt

23.6.2.8. Enum `system_interrupt_vector_saml21`

Table of all possible interrupt and exception vector indexes within the SAM L21 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-22. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_SYSTEM	Interrupt vector index for MCLK, OSCCTRL, OSC32KCTRL, PAC, PM, SUPC, TAL peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_USB	Interrupt vector index for a Universal Serial Bus interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_EVSYN	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCCn	Interrupt vector index for a Timer/Counter Control peripheral interrupt. Each specific device may contain several TCC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TCC0).
SYSTEM_INTERRUPT_MODULE_TCn	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_DAC	Interrupt vector index for a Digital-to-Analog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_PTC	Interrupt vector index for a Peripheral Touch Controller peripheral interrupt
SYSTEM_INTERRUPT_MODULE_AES	Interrupt vector index for a AES peripheral interrupt
SYSTEM_INTERRUPT_MODULE_TRNG	Interrupt vector index for a TRNG peripheral interrupt

23.6.2.9. Enum system_interrupt_vector_saml22

Table of all possible interrupt and exception vector indexes within the SAM L22 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-23. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_SYSTEM	Interrupt vector index for MCLK, OSCCTRL, OSC32KCTRL, PAC, PM, SUPC, TAL peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_FREQM	Interrupt vector index for Frequency Meter peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt
SYSTEM_INTERRUPT_MODULE_EVSYS	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TcN	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).

Enum value	Description
SYSTEM_INTERRUPT_MODULE_TCC0	Interrupt vector index for a Timer/Counter Control peripheral interrupt
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_SLCD	Interrupt vector index for a Peripheral Touch Controller peripheral interrupt

23.6.2.10. Enum system_interrupt_vector_samr21

Table of all possible interrupt and exception vector indexes within the SAM R21 device.

Note: The actual enumeration name is "system_interrupt_vector".

Table 23-24. Members

Enum value	Description
SYSTEM_INTERRUPT_NON_MASKABLE	Interrupt vector index for a NMI interrupt
SYSTEM_INTERRUPT_HARD_FAULT	Interrupt vector index for a Hard Fault memory access exception
SYSTEM_INTERRUPT_SV_CALL	Interrupt vector index for a Supervisor Call exception
SYSTEM_INTERRUPT_PENDING_SV	Interrupt vector index for a Pending Supervisor interrupt
SYSTEM_INTERRUPT_SYSTICK	Interrupt vector index for a System Tick interrupt
SYSTEM_INTERRUPT_MODULE_PM	Interrupt vector index for a Power Manager peripheral interrupt
SYSTEM_INTERRUPT_MODULE_SYSCTRL	Interrupt vector index for a System Control peripheral interrupt
SYSTEM_INTERRUPT_MODULE_WDT	Interrupt vector index for a Watch Dog peripheral interrupt
SYSTEM_INTERRUPT_MODULE_RTC	Interrupt vector index for a Real Time Clock peripheral interrupt
SYSTEM_INTERRUPT_MODULE_EIC	Interrupt vector index for an External Interrupt peripheral interrupt
SYSTEM_INTERRUPT_MODULE_NVMCTRL	Interrupt vector index for a Non Volatile Memory Controller interrupt
SYSTEM_INTERRUPT_MODULE_DMA	Interrupt vector index for a Direct Memory Access interrupt

Enum value	Description
SYSTEM_INTERRUPT_MODULE_USB	Interrupt vector index for a Universal Serial Bus interrupt
SYSTEM_INTERRUPT_MODULE_EVSY	Interrupt vector index for an Event System interrupt
SYSTEM_INTERRUPT_MODULE_SERCOMn	Interrupt vector index for a SERCOM peripheral interrupt. Each specific device may contain several SERCOM peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_SERCOM0).
SYSTEM_INTERRUPT_MODULE_TCCn	Interrupt vector index for a Timer/Counter Control peripheral interrupt. Each specific device may contain several TCC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TCC0).
SYSTEM_INTERRUPT_MODULE_TCn	Interrupt vector index for a Timer/Counter peripheral interrupt. Each specific device may contain several TC peripherals; each module instance will have its own entry in the table, with the instance number substituted for "n" in the entry name (e.g. SYSTEM_INTERRUPT_MODULE_TC3).
SYSTEM_INTERRUPT_MODULE_AC	Interrupt vector index for an Analog Comparator peripheral interrupt
SYSTEM_INTERRUPT_MODULE_ADC	Interrupt vector index for an Analog-to-Digital peripheral interrupt
SYSTEM_INTERRUPT_MODULE_PTC	Interrupt vector index for a Peripheral Touch Controller peripheral interrupt

23.7. Extra Information for SYSTEM INTERRUPT Driver

23.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
ISR	Interrupt Service Routine
NMI	Non-maskable Interrupt
SERCOM	Serial Communication Interface

23.7.2. Dependencies

This driver has the following dependencies:

- None

23.7.3. Errata

There are no errata related to this driver.

23.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

23.8. Examples for SYSTEM INTERRUPT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM System Interrupt \(SYSTEM INTERRUPT\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for SYSTEM INTERRUPT - Critical Section Use Case](#)
- [Quick Start Guide for SYSTEM INTERRUPT - Enable Module Interrupt Use Case](#)

23.8.1. Quick Start Guide for SYSTEM INTERRUPT - Critical Section Use Case

In this case we perform a critical piece of code, disabling all interrupts while a global shared flag is read. During the critical section, no interrupts may occur.

23.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

23.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enter_critical_section();
if (is_ready == true) {
```



```

    /* Do something in response to the global shared flag */
    is_ready = false;
}

system_interrupt_leave_critical_section();

```

Workflow

1. Enter a critical section to disable global interrupts.

```
system_interrupt_enter_critical_section();
```

Note: Critical sections *may* be nested if desired; if nested, global interrupts will only be re-enabled once the outer-most critical section has completed.

2. Check a global shared flag and perform a response. This code may be any critical code that requires exclusive access to all resources without the possibility of interruption.

```

if (is_ready == true) {
    /* Do something in response to the global shared flag */
    is_ready = false;
}

```

3. Exit the critical section to re-enable global interrupts.

```
system_interrupt_leave_critical_section();
```

23.8.2. Quick Start Guide for SYSTEM INTERRUPT - Enable Module Interrupt Use Case

In this case we enable interrupt handling for a specific module, as well as enable interrupts globally for the device.

23.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

23.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```

system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_RTC);
system_interrupt_enable_global();

```

Workflow

1. Enable interrupt handling for the device's RTC peripheral.

```
system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_RTC);
```

2. Enable global interrupts, so that any enabled and active interrupt sources can trigger their respective handler functions.

```
system_interrupt_enable_global();
```

24. SAM System Pin Multiplexer (SYSTEM PINMUX) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's physical I/O Pins, to alter the direction and input/drive characteristics as well as to configure the pin peripheral multiplexer selection.

The following peripheral is used by this module:

- PORT (Port I/O Management)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

24.1. Prerequisites

There are no prerequisites for this module.

24.2. Module Overview

The SAM devices contain a number of General Purpose I/O pins, used to interface the user application logic and internal hardware peripherals to an external system. The Pin Multiplexer (PINMUX) driver provides a method of configuring the individual pin peripheral multiplexers to select alternate pin functions.

24.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_SYSTEM_PINMUX_DRIVE_STRENGTH	SAM L21, SAM C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

24.2.2. Physical and Logical GPIO Pins

SAM devices use two naming conventions for the I/O pins in the device; one physical and one logical. Each physical pin on a device package is assigned both a physical port and pin identifier (e.g. "PORTA.

0") as well as a monotonically incrementing logical GPIO number (e.g. "GPIO0"). While the former is used to map physical pins to their physical internal device module counterparts, for simplicity the design of this driver uses the logical GPIO numbers instead.

24.2.3. Peripheral Multiplexing

SAM devices contain a peripheral MUX, which is individually controllable for each I/O pin of the device. The peripheral MUX allows you to select the function of a physical package pin - whether it will be controlled as a user controllable GPIO pin, or whether it will be connected internally to one of several peripheral modules (such as an I²C module). When a pin is configured in GPIO mode, other peripherals connected to the same pin will be disabled.

24.2.4. Special Pad Characteristics

There are several special modes that can be selected on one or more I/O pins of the device, which alter the input and output characteristics of the pad.

24.2.4.1. Drive Strength

The Drive Strength configures the strength of the output driver on the pad. Normally, there is a fixed current limit that each I/O pin can safely drive, however some I/O pads offer a higher drive mode which increases this limit for that I/O pin at the expense of an increased power consumption.

24.2.4.2. Slew Rate

The Slew Rate configures the slew rate of the output driver, limiting the rate at which the pad output voltage can change with time.

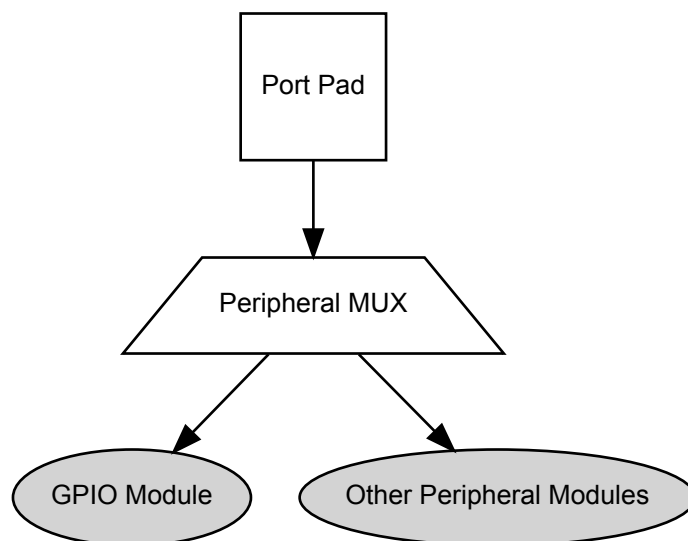
24.2.4.3. Input Sample Mode

The Input Sample Mode configures the input sampler buffer of the pad. By default, the input buffer is only sampled "on-demand", i.e. when the user application attempts to read from the input buffer. This mode is the most power efficient, but increases the latency of the input sample by two clock cycles of the port clock. To reduce latency, the input sampler can instead be configured to always sample the input buffer on each port clock cycle, at the expense of an increased power consumption.

24.2.5. Physical Connection

Figure 24-1 shows how this module is interconnected within the device:

Figure 24-1. Physical Connection



24.3. Special Considerations

The SAM port pin input sampling mode is set in groups of four physical pins; setting the sampling mode of any pin in a sub-group of eight I/O pins will configure the sampling mode of the entire sub-group.

High Drive Strength output driver mode is not available on all device pins - refer to your device specific datasheet.

24.4. Extra Information

For extra information, see [Extra Information for SYSTEM PINMUX Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

24.5. Examples

For a list of examples related to this driver, see [Examples for SYSTEM PINMUX Driver](#).

24.6. API Overview

24.6.1. Structure Definitions

24.6.1.1. Struct `system_pinmux_config`

Configuration structure for a port pin instance. This structure should be initialized by the [system_pinmux_get_config_defaults\(\)](#) function before being modified by the user application.

Table 24-1. Members

Type	Name	Description
enum system_pinmux_pin_dir	direction	Port buffer input/output direction
enum system_pinmux_pin_pull	input_pull	Logic level pull of the input buffer
uint8_t	mux_position	MUX index of the peripheral that should control the pin, if peripheral control is desired. For GPIO use, this should be set to SYSTEM_PINMUX_GPIO .
bool	powersave	Enable lowest possible powerstate on the pin Note: All other configurations will be ignored, the pin will be disabled.

24.6.2. Macro Definitions

24.6.2.1. Macro FEATURE_SYSTEM_PINMUX_DRIVE_STRENGTH

```
#define FEATURE_SYSTEM_PINMUX_DRIVE_STRENGTH
```

Output Driver Strength Selection feature support

24.6.2.2. Macro SYSTEM_PINMUX_GPIO

```
#define SYSTEM_PINMUX_GPIO
```

Peripheral multiplexer index to select GPIO mode for a pin

24.6.3. Function Definitions

24.6.3.1. Configuration and Initialization

Function `system_pinmux_get_config_defaults()`

Initializes a Port pin configuration structure to defaults.

```
void system_pinmux_get_config_defaults(  
    struct system_pinmux_config *const config)
```

Initializes a given Port pin configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Non peripheral (i.e. GPIO) controlled
- Input mode with internal pull-up enabled

Table 24-2. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `system_pinmux_pin_set_config()`

Writes a Port pin configuration to the hardware module.

```
void system_pinmux_pin_set_config(  
    const uint8_t gpio_pin,  
    const struct system_pinmux_config *const config)
```

Writes out a given configuration of a Port pin configuration to the hardware module.

Note: If the pin direction is set as an output, the pull-up/pull-down input configuration setting is ignored.

Table 24-3. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure
[in]	config	Configuration settings for the pin

Function `system_pinmux_group_set_config()`

Writes a Port pin group configuration to the hardware module.

```
void system_pinmux_group_set_config(  
    PortGroup *const port,  
    const uint32_t mask,  
    const struct system_pinmux_config *const config)
```

Writes out a given configuration of a Port pin group configuration to the hardware module.

Note: If the pin direction is set as an output, the pull-up/pull-down input configuration setting is ignored.

Table 24-4. Parameters

Data direction	Parameter name	Description
[in]	port	Base of the PORT module to configure
[in]	mask	Mask of the port pin(s) to configure
[in]	config	Configuration settings for the pin

24.6.3.2. Special Mode Configuration (Physical Group Orientated)

Function `system_pinmux_get_group_from_gpio_pin()`

Retrieves the PORT module group instance from a given GPIO pin number.

```
PortGroup * system_pinmux_get_group_from_gpio_pin(  
    const uint8_t gpio_pin)
```

Retrieves the PORT module group instance associated with a given logical GPIO pin number.

Table 24-5. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to convert

Returns

Base address of the associated PORT module.

Function `system_pinmux_group_set_input_sample_mode()`

Configures the input sampling mode for a group of pins.

```
void system_pinmux_group_set_input_sample_mode(  
    PortGroup *const port,  
    const uint32_t mask,  
    const enum system_pinmux_pin_sample mode)
```

Configures the input sampling mode for a group of pins, to control when the physical I/O pin value is sampled and stored inside the microcontroller.

Table 24-6. Parameters

Data direction	Parameter name	Description
[in]	port	Base of the PORT module to configure
[in]	mask	Mask of the port pin(s) to configure
[in]	mode	New pin sampling mode to configure

24.6.3.3. Special Mode Configuration (Logical Pin Orientated)

Function `system_pinmux_pin_get_mux_position()`

Retrieves the currently selected MUX position of a logical pin.

```
uint8_t system_pinmux_pin_get_mux_position(  
    const uint8_t gpio_pin)
```

Retrieves the selected MUX peripheral on a given logical GPIO pin.

Table 24-7. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure

Returns

Currently selected peripheral index on the specified pin.

Function `system_pinmux_pin_set_input_sample_mode()`

Configures the input sampling mode for a GPIO pin.

```
void system_pinmux_pin_set_input_sample_mode(  
    const uint8_t gpio_pin,  
    const enum system_pinmux_pin_sample mode)
```

Configures the input sampling mode for a GPIO input, to control when the physical I/O pin value is sampled and stored inside the microcontroller.

Table 24-8. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure
[in]	mode	New pin sampling mode to configure

24.6.3.4. Function `system_pinmux_group_set_output_strength()`

Configures the output driver strength mode for a group of pins.

```
void system_pinmux_group_set_output_strength(  
    PortGroup *const port,  
    const uint32_t mask,  
    const enum system_pinmux_pin_strength mode)
```

Configures the output drive strength for a group of pins, to control the amount of current the pad is able to sink/source.

Table 24-9. Parameters

Data direction	Parameter name	Description
[in]	port	Base of the PORT module to configure
[in]	mask	Mask of the port pin(s) to configure
[in]	mode	New output driver strength mode to configure

24.6.3.5. Function `system_pinmux_pin_set_output_strength()`

Configures the output driver strength mode for a GPIO pin.

```
void system_pinmux_pin_set_output_strength(  
    const uint8_t gpio_pin,  
    const enum system_pinmux_pin_strength mode)
```

Configures the output drive strength for a GPIO output, to control the amount of current the pad is able to sink/source.

Table 24-10. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure
[in]	mode	New output driver strength mode to configure

24.6.4. Enumeration Definitions

24.6.4.1. Enum `system_pinmux_pin_dir`

Enum for the possible pin direction settings of the port pin configuration structure, to indicate the direction the pin should use.

Table 24-11. Members

Enum value	Description
SYSTEM_PINMUX_PIN_DIR_INPUT	The pin's input buffer should be enabled, so that the pin state can be read
SYSTEM_PINMUX_PIN_DIR_OUTPUT	The pin's output buffer should be enabled, so that the pin state can be set (but not read back)
SYSTEM_PINMUX_PIN_DIR_OUTPUT_WITH_READBACK	The pin's output and input buffers should both be enabled, so that the pin state can be set and read back

24.6.4.2. Enum system_pinmux_pin_pull

Enum for the possible pin pull settings of the port pin configuration structure, to indicate the type of logic level pull the pin should use.

Table 24-12. Members

Enum value	Description
SYSTEM_PINMUX_PIN_PULL_NONE	No logical pull should be applied to the pin
SYSTEM_PINMUX_PIN_PULL_UP	Pin should be pulled up when idle
SYSTEM_PINMUX_PIN_PULL_DOWN	Pin should be pulled down when idle

24.6.4.3. Enum system_pinmux_pin_sample

Enum for the possible input sampling modes for the port pin configuration structure, to indicate the type of sampling a port pin should use.

Table 24-13. Members

Enum value	Description
SYSTEM_PINMUX_PIN_SAMPLE_CONTINUOUS	Pin input buffer should continuously sample the pin state
SYSTEM_PINMUX_PIN_SAMPLE_ONDEMAND	Pin input buffer should be enabled when the IN register is read

24.6.4.4. Enum system_pinmux_pin_strength

Enum for the possible output drive strengths for the port pin configuration structure, to indicate the driver strength the pin should use.

Table 24-14. Members

Enum value	Description
SYSTEM_PINMUX_PIN_STRENGTH_NORMAL	Normal output driver strength
SYSTEM_PINMUX_PIN_STRENGTH_HIGH	High current output driver strength

24.7. Extra Information for SYSTEM PINMUX Driver

24.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
GPIO	General Purpose Input/Output
MUX	Multiplexer

24.7.2. Dependencies

This driver has the following dependencies:

- None

24.7.3. Errata

There are no errata related to this driver.

24.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Removed code of open drain, slew limit and drive strength features
Fixed broken sampling mode function implementations, which wrote corrupt configuration values to the device registers
Added missing NULL pointer asserts to the PORT driver functions
Initial Release

24.8. Examples for SYSTEM PINMUX Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM System Pin Multiplexer \(SYSTEM PINMUX\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for SYSTEM PINMUX - Basic](#)

24.8.1. Quick Start Guide for SYSTEM PINMUX - Basic

In this use case, the PINMUX module is configured for:

- One pin in input mode, with pull-up enabled, connected to the GPIO module
- Sampling mode of the pin changed to sample on demand

This use case sets up the PINMUX to configure a physical I/O pin set as an input with pull-up and changes the sampling mode of the pin to reduce power by only sampling the physical pin state when the user application attempts to read it.

24.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your application:

```
struct system_pinmux_config config_pinmux;
system_pinmux_get_config_defaults(&config_pinmux);

config_pinmux.mux_position = SYSTEM_PINMUX_GPIO;
config_pinmux.direction    = SYSTEM_PINMUX_PIN_DIR_INPUT;
config_pinmux.input_pull   = SYSTEM_PINMUX_PIN_PULL_UP;

system_pinmux_pin_set_config(10, &config_pinmux);
```

Workflow

1. Create a PINMUX module pin configuration struct, which can be filled out to adjust the configuration of a single port pin.

```
struct system_pinmux_config config_pinmux;
```

2. Initialize the pin configuration struct with the module's default values.

```
system_pinmux_get_config_defaults(&config_pinmux);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to request an input pin with pull-up connected to the GPIO peripheral.

```
config_pinmux.mux_position = SYSTEM_PINMUX_GPIO;
config_pinmux.direction    = SYSTEM_PINMUX_PIN_DIR_INPUT;
config_pinmux.input_pull   = SYSTEM_PINMUX_PIN_PULL_UP;
```

4. Configure GPIO10 with the initialized pin configuration struct, to enable the input sampler on the pin.

```
system_pinmux_pin_set_config(10, &config_pinmux);
```

24.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_pinmux_pin_set_input_sample_mode(10,
    SYSTEM_PINMUX_PIN_SAMPLE_ONDEMAND);

while (true) {
    /* Infinite loop */
}
```

Workflow

1. Adjust the configuration of the pin to enable on-demand sampling mode.

```
system_pinmux_pin_set_input_sample_mode(10,  
    SYSTEM_PINMUX_PIN_SAMPLE_ONDEMAND);
```

25. SAM Timer/Counter (TC) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the timer modules within the device, for waveform generation and timing operations. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- TC (Timer/Counter)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

25.1. Prerequisites

There are no prerequisites for this module.

25.2. Module Overview

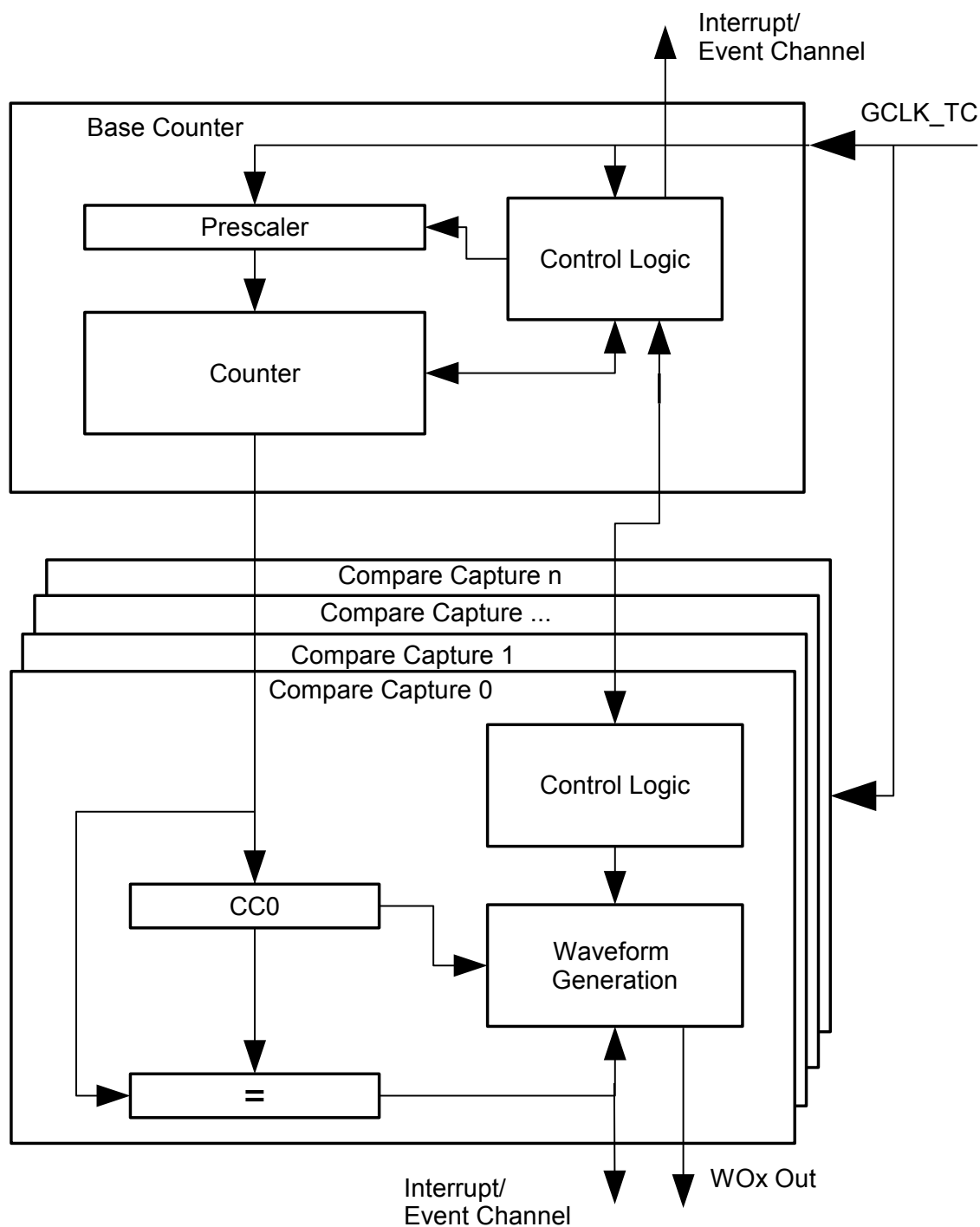
The Timer/Counter (TC) module provides a set of timing and counting related functionality, such as the generation of periodic waveforms, the capturing of a periodic waveform's frequency/duty cycle, and software timekeeping for periodic operations. TC modules can be configured to use an 8-, 16-, or 32-bit counter size.

This TC module for the SAM is capable of the following functions:

- Generation of PWM signals
- Generation of timestamps for events
- General time counting
- Waveform period capture
- Waveform frequency capture

[Figure 25-1](#) shows the overview of the TC module design.

Figure 25-1. Basic Overview of the TC Module



25.2.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_TC_DOUBLE_BUFFERED	SAM L21/L22/C20/C21
FEATURE_TC_SYNCBUSY_SCHEME_VERSION_2	SAM L21/L22/C20/C21
FEATURE_TC_STAMP_PW_CAPTURE	SAM L21/L22/C20/C21
FEATURE_TC_READ_SYNC	SAM L21/L22/C20/C21
FEATURE_TC_IO_CAPTURE	SAM L21/L22/C20/C21
FEATURE_TC_GENERATE_DMA_TRIGGER	SAM L21/L22

Note: The specific features are only available in the driver when the selected device supports those features.

25.2.2. Functional Description

Independent of the configured counter size, each TC module can be set up in one of two different modes; capture and compare.

In capture mode, the counter value is stored when a configurable event occurs. This mode can be used to generate timestamps used in event capture, or it can be used for the measurement of a periodic input signal's frequency/duty cycle.

In compare mode, the counter value is compared against one or more of the configured channel compare values. When the counter value coincides with a compare value an action can be taken automatically by the module, such as generating an output event or toggling a pin when used for frequency or Pulse Width Modulation (PWM) signal generation.

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the input event of another. For more information on event routing, refer to the event driver documentation.

25.2.3. Timer/Counter Size

Each timer module can be configured in one of three different counter sizes; 8-, 16-, and 32-bit. The size of the counter determines the maximum value it can count to before an overflow occurs and the count is reset back to zero. [Table 25-1](#) shows the maximum values for each of the possible counter sizes.

Table 25-1. Timer Counter Sizes and Their Maximum Count Values

Counter size	Max. (hexadecimal)	Max. (decimal)
8-bit	0xFF	255
16-bit	0xFFFF	65,535
32-bit	0xFFFFFFFF	4,294,967,295

When using the counter in 16- or 32-bit count mode, Compare Capture register 0 (CC0) is used to store the period value when running in PWM generation match mode.

When using 32-bit counter size, two 16-bit counters are chained together in a cascade formation. Except in SAM D09/D10/D11. Even numbered TC modules (e.g. TC0, TC2) can be configured as 32-bit counters. The odd numbered counters will act as slaves to the even numbered masters, and will not be reconfigurable until the master timer is disabled. The pairing of timer modules for 32-bit mode is shown in [Table 25-2](#).

Table 25-2. TC Master and Slave Module Pairings

Master TC module	Slave TC module
TC0	TC1
TC2	TC3
...	...
TCn-1	TCn

In SAM D09/D10/D11, odd numbered TC modules (e.g. TC1) can be configured as 32-bit counters. The even numbered (e.g. TC2) counters will act as slaves to the odd numbered masters.

25.2.4. Clock Settings

25.2.4.1. Clock Selection

Each TC peripheral is clocked asynchronously to the system clock by a GCLK (Generic Clock) channel. The GCLK channel connects to any of the GCLK generators. The GCLK generators are configured to use one of the available clock sources on the system such as internal oscillator, external crystals, etc. See the Generic Clock driver for more information.

25.2.4.2. Prescaler

Each TC module in the SAM has its own individual clock prescaler, which can be used to divide the input clock frequency used in the counter. This prescaler only scales the clock used to provide clock pulses for the counter to count, and does not affect the digital register interface portion of the module, thus the timer registers will synchronize to the raw GCLK frequency input to the module.

As a result of this, when selecting a GCLK frequency and timer prescaler value the user application should consider both the timer resolution required and the synchronization frequency, to avoid lengthy synchronization times of the module if a very slow GCLK frequency is fed into the TC module. It is preferable to use a higher module GCLK frequency as the input to the timer, and prescale this down as much as possible to obtain a suitable counter frequency in latency-sensitive applications.

25.2.4.3. Reloading

Timer modules also contain a configurable reload action, used when a re-trigger event occurs. Examples of a re-trigger event are the counter reaching the maximum value when counting up, or when an event from the event system tells the counter to re-trigger. The reload action determines if the prescaler should be reset, and when this should happen. The counter will always be reloaded with the value it is set to start counting from. The user can choose between three different reload actions, described in [Table 25-3](#).

Table 25-3. TC Module Reload Actions

Reload action	Description
TC_RELOAD_ACTION_GCLK	Reload TC counter value on next GCLK cycle. Leave prescaler as-is.
TC_RELOAD_ACTION_PRESC	Reloads TC counter value on next prescaler clock. Leave prescaler as-is.
TC_RELOAD_ACTION_RESYNC	Reload TC counter value on next GCLK cycle. Clear prescaler to zero.

The reload action to use will depend on the specific application being implemented. One example is when an external trigger for a reload occurs; if the TC uses the prescaler, the counter in the prescaler should not have a value between zero and the division factor. The TC counter and the counter in the prescaler should both start at zero. When the counter is set to re-trigger when it reaches the maximum value on the other hand, this is not the right option to use. In such a case it would be better if the prescaler is left unaltered when the re-trigger happens, letting the counter reset on the next GCLK cycle.

25.2.5. Compare Match Operations

In compare match operation, Compare/Capture registers are used in comparison with the counter value. When the timer's count value matches the value of a compare channel, a user defined action can be taken.

25.2.5.1. Basic Timer

A Basic Timer is a simple application where compare match operations are used to determine when a specific period has elapsed. In Basic Timer operations, one or more values in the module's Compare/Capture registers are used to specify the time (as a number of prescaled GCLK cycles) when an action should be taken by the microcontroller. This can be an Interrupt Service Routine (ISR), event generator via the event system, or a software flag that is polled via the user application.

25.2.5.2. Waveform Generation

Waveform generation enables the TC module to generate square waves, or if combined with an external passive low-pass filter; analog waveforms.

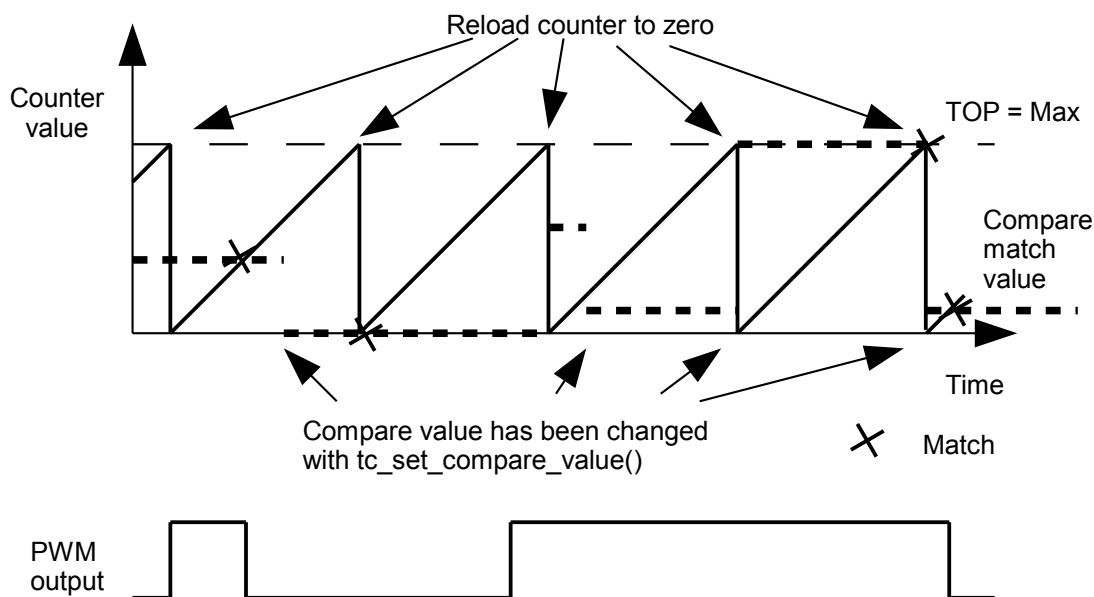
25.2.5.3. Waveform Generation - PWM

Pulse width modulation is a form of waveform generation and a signalling technique that can be useful in many situations. When PWM mode is used, a digital pulse train with a configurable frequency and duty cycle can be generated by the TC module and output to a GPIO pin of the device.

Often PWM is used to communicate a control or information parameter to an external circuit or component. Differing impedances of the source generator and sink receiver circuits are less of an issue when using PWM compared to using an analog voltage value, as noise will not generally affect the signal's integrity to a meaningful extent.

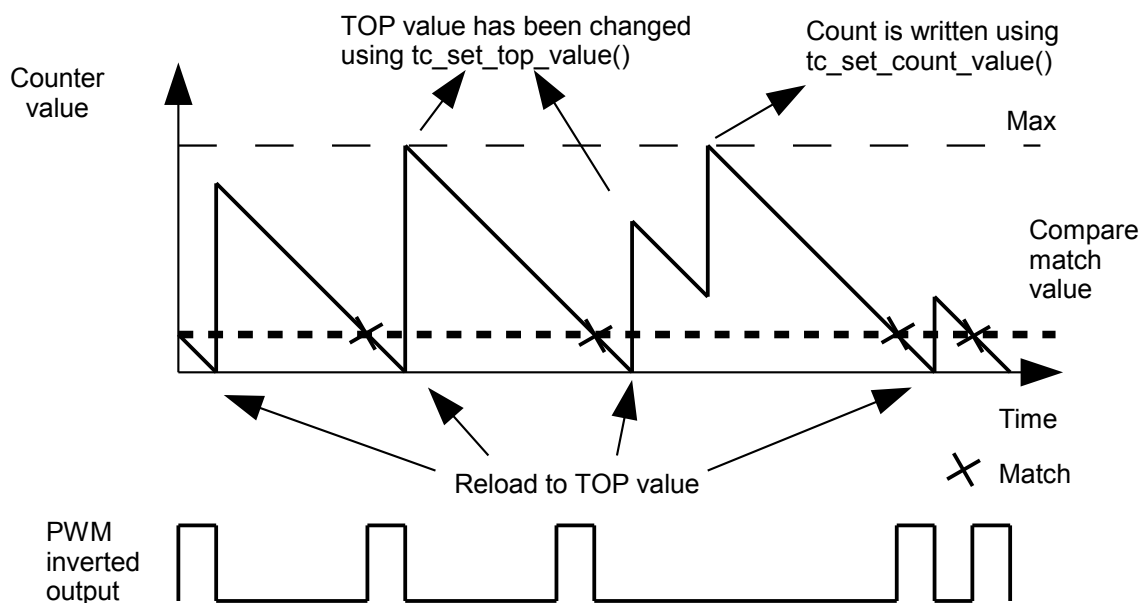
[Figure 25-2](#) illustrates operations and different states of the counter and its output when running the counter in PWM normal mode. As can be seen, the TOP value is unchanged and is set to MAX. The compare match value is changed at several points to illustrate the resulting waveform output changes. The PWM output is set to normal (i.e. non-inverted) output mode.

Figure 25-2. Example of PWM in Normal Mode, and Different Counter Operations



In [Figure 25-3](#), the counter is set to generate PWM in Match mode. The PWM output is inverted via the appropriate configuration option in the TC driver configuration structure. In this example, the counter value is changed once, but the compare match value is kept unchanged. As can be seen, it is possible to change the TOP value when running in PWM match mode.

Figure 25-3. Example of PWM in Match Mode and Different Counter Operations



25.2.5.4. Waveform Generation - Frequency

Frequency Generation mode is in many ways identical to PWM generation. However, in Frequency Generation a toggle only occurs on the output when a match on a capture channels occurs. When the

match is made, the timer value is reset, resulting in a variable frequency square wave with a fixed 50% duty cycle.

25.2.5.5. Capture Operations

In capture operations, any event from the event system or a pin change can trigger a capture of the counter value. This captured counter value can be used as a timestamp for the event, or it can be used in frequency and pulse width capture.

25.2.5.6. Capture Operations - Event

Event capture is a simple use of the capture functionality, designed to create timestamps for specific events. When the TC module's input capture pin is externally toggled, the current timer count value is copied into a buffered register which can then be read out by the user application.

Note that when performing any capture operation, there is a risk that the counter reaches its top value (MAX) when counting up, or the bottom value (zero) when counting down, before the capture event occurs. This can distort the result, making event timestamps to appear shorter than reality; the user application should check for timer overflow when reading a capture result in order to detect this situation and perform an appropriate adjustment.

Before checking for a new capture, [TC_STATUS_COUNT_OVERFLOW](#) should be checked. The response to an overflow error is left to the user application, however it may be necessary to clear both the capture overflow flag and the capture flag upon each capture reading.

25.2.5.7. Capture Operations - Pulse Width

Pulse Width Capture mode makes it possible to measure the pulse width and period of PWM signals. This mode uses two capture channels of the counter. This means that the counter module used for Pulse Width Capture can not be used for any other purpose. There are two modes for pulse width capture; Pulse Width Period (PWP) and Period Pulse Width (PPW). In PWP mode, capture channel 0 is used for storing the pulse width and capture channel 1 stores the observed period. While in PPW mode, the roles of the two capture channels are reversed.

As in the above example it is necessary to poll on interrupt flags to see if a new capture has happened and check that a capture overflow error has not occurred.

25.2.6. One-shot Mode

TC modules can be configured into a one-shot mode. When configured in this manner, starting the timer will cause it to count until the next overflow or underflow condition before automatically halting, waiting to be manually triggered by the user application software or an event signal from the event system.

25.2.6.1. Wave Generation Output Inversion

The output of the wave generation can be inverted by hardware if desired, resulting in the logically inverted value being output to the configured device GPIO pin.

25.3. Special Considerations

The number of capture compare registers in each TC module is dependent on the specific SAM device being used, and in some cases the counter size.

The maximum amount of capture compare registers available in any SAM device is two when running in 32-bit mode and four in 8- and 16-bit modes.

25.4. Extra Information

For extra information, see [Extra Information for TC Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

25.5. Examples

For a list of examples related to this driver, see [Examples for TC Driver](#).

25.6. API Overview

25.6.1. Variable and Type Definitions

25.6.1.1. Waveform Inversion Mode

Type `tc_callback_t`

```
typedef void(* tc_callback_t )(struct tc_module *const module)
```

Type of the callback functions.

25.6.2. Structure Definitions

25.6.2.1. Struct `tc_16bit_config`

Table 25-4. Members

Type	Name	Description
uint16_t	compare_capture_channel[]	Value to be used for compare match on each channel
uint16_t	value	Initial timer count value

25.6.2.2. Struct `tc_32bit_config`

Table 25-5. Members

Type	Name	Description
uint32_t	compare_capture_channel[]	Value to be used for compare match on each channel
uint32_t	value	Initial timer count value

25.6.2.3. Struct tc_8bit_config

Table 25-6. Members

Type	Name	Description
uint8_t	compare_capture_channel[]	Value to be used for compare match on each channel
uint8_t	period	Where to count to or from depending on the direction on the counter
uint8_t	value	Initial timer count value

25.6.2.4. Struct tc_config

Configuration struct for a TC instance. This structure should be initialized by the [tc_get_config_defaults](#) function before being modified by the user application.

Table 25-7. Members

Type	Name	Description
union tc_config.@1	@1	Access the different counter size settings through this configuration member.
enum tc_clock_prescaler	clock_prescaler	Specifies the prescaler value for GCLK_TC
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
enum tc_count_direction	count_direction	Specifies the direction for the TC to count
enum tc_counter_size	counter_size	Specifies either 8-, 16-, or 32-bit counter size
bool	double_buffering_enabled	Set to <code>true</code> to enable double buffering write. When enabled any write through tc_set_top_value() , tc_set_compare_value() and will direct to the buffer register as buffered value, and the buffered value will be committed to effective register on UPDATE condition, if update is not locked.
bool	enable_capture_on_channel[]	Specifies which channel(s) to enable channel capture operation on
bool	enable_capture_on_IO[]	Specifies which channel(s) to enable I/O capture operation on
bool	oneshot	When <code>true</code> , one-shot will stop the TC on next hardware or software re-trigger event or overflow/underflow
struct tc_pwm_channel	pwm_channel[]	Specifies the PWM channel for TC

Type	Name	Description
enum tc_reload_action	reload_action	Specifies the reload or reset time of the counter and prescaler resynchronization on a re-trigger event for the TC
bool	run_in_standby	When <code>true</code> the module is enabled during standby
enum tc_wave_generation	wave_generation	Specifies which waveform generation mode to use
uint8_t	waveform_invert_output	Specifies which channel(s) to invert the waveform on. For SAM L21/L22/C20/C21, it's also used to invert I/O input pin.

25.6.2.5. Union `tc_config.__unnamed__`

Access the different counter size settings through this configuration member.

Table 25-8. Members

Type	Name	Description
struct tc_16bit_config	counter_16_bit	Struct for 16-bit specific timer configuration
struct tc_32bit_config	counter_32_bit	Struct for 32-bit specific timer configuration
struct tc_8bit_config	counter_8_bit	Struct for 8-bit specific timer configuration

25.6.2.6. Struct `tc_events`

Event flags for the [tc_enable_events\(\)](#) and [tc_disable_events\(\)](#).

Table 25-9. Members

Type	Name	Description
enum tc_event_action	event_action	Specifies which event to trigger if an event is triggered
bool	generate_event_on_compare_channel[]	Generate an output event on a compare channel match
bool	generate_event_on_overflow	Generate an output event on counter overflow
bool	invert_event_input	Specifies if the input event source is inverted, when used in PWP or PPW event action modes
bool	on_event_perform_action	Perform the configured event action when an incoming event is signalled

25.6.2.7. Struct tc_module

TC software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

25.6.2.8. Struct tc_pwm_channel

Table 25-10. Members

Type	Name	Description
bool	enabled	When true, PWM output for the given channel is enabled
uint32_t	pin_mux	Specifies Multiplexer (MUX) setting for each output channel pin
uint32_t	pin_out	Specifies pin output for each channel

25.6.3. Macro Definitions

25.6.3.1. Macro FEATURE_TC_DOUBLE_BUFFERED

```
#define FEATURE_TC_DOUBLE_BUFFERED
```

Define port features set according to different device familyTC double buffered.

25.6.3.2. Macro FEATURE_TC_SYNCBUSY_SCHEME_VERSION_2

```
#define FEATURE_TC_SYNCBUSY_SCHEME_VERSION_2
```

SYNCBUSY scheme version 2.

25.6.3.3. Macro FEATURE_TC_STAMP_PW_CAPTURE

```
#define FEATURE_TC_STAMP_PW_CAPTURE
```

TC time stamp capture and pulse width capture.

25.6.3.4. Macro FEATURE_TC_READ_SYNC

```
#define FEATURE_TC_READ_SYNC
```

Read synchronization of COUNT.

25.6.3.5. Macro FEATURE_TC_IO_CAPTURE

```
#define FEATURE_TC_IO_CAPTURE
```

I/O pin edge capture.

25.6.3.6. Macro FEATURE_TC_GENERATE_DMA_TRIGGER

```
#define FEATURE_TC_GENERATE_DMA_TRIGGER
```

Generate Direct Memory Access (DMA) triggers.

25.6.3.7. Module Status Flags

TC status flags, returned by [tc_get_status\(\)](#) and cleared by [tc_clear_status\(\)](#).

Macro TC_STATUS_CHANNEL_0_MATCH

```
#define TC_STATUS_CHANNEL_0_MATCH
```

Timer channel 0 has matched against its compare value, or has captured a new value.

Macro TC_STATUS_CHANNEL_1_MATCH

```
#define TC_STATUS_CHANNEL_1_MATCH
```

Timer channel 1 has matched against its compare value, or has captured a new value.

Macro TC_STATUS_SYNC_READY

```
#define TC_STATUS_SYNC_READY
```

Timer register synchronization has completed, and the synchronized count value may be read.

Macro TC_STATUS_CAPTURE_OVERFLOW

```
#define TC_STATUS_CAPTURE_OVERFLOW
```

A new value was captured before the previous value was read, resulting in lost data.

Macro TC_STATUS_COUNT_OVERFLOW

```
#define TC_STATUS_COUNT_OVERFLOW
```

The timer count value has overflowed from its maximum value to its minimum when counting upward, or from its minimum value to its maximum when counting downward.

Macro TC_STATUS_CHN0_BUFFER_VALID

```
#define TC_STATUS_CHN0_BUFFER_VALID
```

Channel 0 compare or capture buffer valid.

Macro TC_STATUS_CHN1_BUFFER_VALID

```
#define TC_STATUS_CHN1_BUFFER_VALID
```

Channel 1 compare or capture buffer valid.

Macro TC_STATUS_PERIOD_BUFFER_VALID

```
#define TC_STATUS_PERIOD_BUFFER_VALID
```

Period buffer valid.

25.6.3.8. TC Wave Generation Mode

Macro TC_WAVE_GENERATION_NORMAL_FREQ_MODE

```
#define TC_WAVE_GENERATION_NORMAL_FREQ_MODE
```

TC wave generation mode: normal frequency.

Macro TC_WAVE_GENERATION_MATCH_FREQ_MODE

```
#define TC_WAVE_GENERATION_MATCH_FREQ_MODE
```

TC wave generation mode: match frequency.

Macro TC_WAVE_GENERATION_NORMAL_PWM_MODE

```
#define TC_WAVE_GENERATION_NORMAL_PWM_MODE
```

TC wave generation mode: normal PWM.

Macro TC_WAVE_GENERATION_MATCH_PWM_MODE

```
#define TC_WAVE_GENERATION_MATCH_PWM_MODE
```

TC wave generation mode: match PWM.

25.6.3.9. Waveform Inversion Mode

Macro TC_WAVEFORM_INVERT_CC0_MODE

```
#define TC_WAVEFORM_INVERT_CC0_MODE
```

Waveform inversion CC0 mode.

Macro TC_WAVEFORM_INVERT_CC1_MODE

```
#define TC_WAVEFORM_INVERT_CC1_MODE
```

Waveform inversion CC1 mode.

25.6.4. Function Definitions

25.6.4.1. Driver Initialization and Configuration

Function tc_is_syncing()

Determines if the hardware module(s) are currently synchronizing to the bus.

```
bool tc_is_syncing(  
    const struct tc_module *const module_inst)
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Table 25-11. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Returns

Synchronization status of the underlying hardware module(s).

Table 25-12. Return Values

Return value	Description
false	If the module has completed synchronization
true	If the module synchronization is ongoing

Function `tc_get_config_defaults()`

Initializes config with predefined default values.

```
void tc_get_config_defaults(  
    struct tc_config *const config)
```

This function will initialize a given TC configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit counter size on the counter
- No prescaler
- Normal frequency wave generation
- GCLK reload action
- Don't run in standby
- Don't run on demand for SAM L21/L22/C20/C21
- No inversion of waveform output
- No capture enabled
- No I/O capture enabled for SAM L21/L22/C20/C21
- No event input enabled
- Count upward
- Don't perform one-shot operations
- No event action
- No channel 0 PWM output
- No channel 1 PWM output
- Counter starts on 0
- Capture compare channel 0 set to 0
- Capture compare channel 1 set to 0
- No PWM pin output enabled
- Pin and MUX configuration not set
- Double buffer disabled (if have this feature)

Table 25-13. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to a TC module configuration structure to set

Function `tc_init()`

Initializes a hardware TC module instance.

```
enum status_code tc_init(  
    struct tc_module *const module_inst,  
    Tc *const hw,  
    const struct tc_config *const config)
```

Enables the clock and initializes the TC module, based on the given configuration values.

Table 25-14. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	hw	Pointer to the TC hardware module
[in]	config	Pointer to the TC configuration options struct

Returns

Status of the initialization procedure.

Table 25-15. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_BUSY	Hardware module was busy when the initialization procedure was attempted
STATUS_INVALID_ARG	An invalid configuration option or argument was supplied
STATUS_ERR_DENIED	Hardware module was already enabled, or the hardware module is configured in 32-bit slave mode

25.6.4.2. Event Management**Function tc_enable_events()**

Enables a TC module event input or output.

```
void tc_enable_events(
    struct tc_module *const module_inst,
    struct tc_events *const events)
```

Enables one or more input or output events to or from the TC module. See [tc_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 25-16. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	events	Struct containing flags of events to enable

Function tc_disable_events()

Disables a TC module event input or output.

```
void tc_disable_events(
    struct tc_module *const module_inst,
    struct tc_events *const events)
```

Disables one or more input or output events to or from the TC module. See [tc_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 25-17. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	events	Struct containing flags of events to disable

25.6.4.3. Enable/Disable/Reset

Function `tc_reset()`

Resets the TC module.

```
enum status_code tc_reset(  
    const struct tc_module *const module_inst)
```

Resets the TC module, restoring all hardware module registers to their default values and disabling the module. The TC module will not be accessible while the reset is being performed.

Note: When resetting a 32-bit counter only the master TC module's instance structure should be passed to the function.

Table 25-18. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Returns

Status of the procedure.

Table 25-19. Return Values

Return value	Description
STATUS_OK	The module was reset successfully
STATUS_ERR_UNSUPPORTED_DEV	A 32-bit slave TC module was passed to the function. Only use reset on master TC

Function `tc_enable()`

Enable the TC module.

```
void tc_enable(  
    const struct tc_module *const module_inst)
```

Enables a TC module that has been previously initialized. The counter will start when the counter is enabled.

Note: When the counter is configured to re-trigger on an event, the counter will not start until the start function is used.

Table 25-20. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Function tc_disable()

Disables the TC module.

```
void tc_disable(
    const struct tc_module *const module_inst)
```

Disables a TC module and stops the counter.

Table 25-21. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

25.6.4.4. Get/Set Count Value**Function tc_get_count_value()**

Get TC module count value.

```
uint32_t tc_get_count_value(
    const struct tc_module *const module_inst)
```

Retrieves the current count value of a TC module. The specified TC module may be started or stopped.

Table 25-22. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Returns

Count value of the specified TC module.

Function tc_set_count_value()

Sets TC module count value.

```
enum status_code tc_set_count_value(
    const struct tc_module *const module_inst,
    const uint32_t count)
```

Sets the current timer count value of a initialized TC module. The specified TC module may be started or stopped.

Table 25-23. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	count	New timer count value to set

Returns

Status of the count update procedure.

Table 25-24. Return Values

Return value	Description
STATUS_OK	The timer count was updated successfully
STATUS_ERR_INVALID_ARG	An invalid timer counter size was specified

25.6.4.5. Start/Stop Counter

Function `tc_stop_counter()`

Stops the counter.

```
void tc_stop_counter(  
    const struct tc_module *const module_inst)
```

This function will stop the counter. When the counter is stopped the value in the count value is set to 0 if the counter was counting up, or maximum if the counter was counting down when stopped.

Table 25-25. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Function `tc_start_counter()`

Starts the counter.

```
void tc_start_counter(  
    const struct tc_module *const module_inst)
```

Starts or restarts an initialized TC module's counter.

Table 25-26. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

25.6.4.6. Double Buffering

Function `tc_update_double_buffer()`

Update double buffer.

```
void tc_update_double_buffer(  
    const struct tc_module *const module_inst)
```

Update double buffer.

Table 25-27. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

25.6.4.7. Count Read Synchronization

Function `tc_sync_read_count()`

Read synchronization of COUNT.

```
void tc_sync_read_count(  
    const struct tc_module *const module_inst)
```

Read synchronization of COUNT.

Table 25-28. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

25.6.4.8. Generate TC DMA Triggers Command

Function `tc_dma_trigger_command()`

TC DMA Trigger.

```
void tc_dma_trigger_command(  
    const struct tc_module *const module_inst)
```

TC DMA trigger command.

Table 25-29. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

25.6.4.9. Get Capture Set Compare

Function `tc_get_capture_value()`

Gets the TC module capture value.

```
uint32_t tc_get_capture_value(  
    const struct tc_module *const module_inst,  
    const enum tc_compare_capture_channel channel_index)
```

Retrieves the capture value in the indicated TC module capture channel.

Table 25-30. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	channel_index	Index of the Compare Capture channel to read

Returns

Capture value stored in the specified timer channel.

Function `tc_set_compare_value()`

Sets a TC module compare value.

```
enum status_code tc_set_compare_value(  
    const struct tc_module *const module_inst,
```

```
const enum tc_compare_capture_channel channel_index,
const uint32_t compare_value)
```

Writes a compare value to the given TC module compare/capture channel.

Table 25-31. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	channel_index	Index of the compare channel to write to
[in]	compare	New compare value to set

Returns

Status of the compare update procedure.

Table 25-32. Return Values

Return value	Description
STATUS_OK	The compare value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied

25.6.4.10. Set Top Value

Function tc_set_top_value()

Set the timer TOP/period value.

```
enum status_code tc_set_top_value(
    const struct tc_module *const module_inst,
    const uint32_t top_value)
```

For 8-bit counter size this function writes the top value to the period register.

For 16- and 32-bit counter size this function writes the top value to Capture Compare register 0. The value in this register can not be used for any other purpose.

Note: This function is designed to be used in PWM or frequency match modes only, when the counter is set to 16- or 32-bit counter size. In 8-bit counter size it will always be possible to change the top value even in normal mode.

Table 25-33. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	top_value	New timer TOP value to set

Returns

Status of the TOP set procedure.

Table 25-34. Return Values

Return value	Description
STATUS_OK	The timer TOP value was updated successfully
STATUS_ERR_INVALID_ARG	The configured TC module counter size in the module instance is invalid

25.6.4.11. Status Management

Function `tc_get_status()`

Retrieves the current module status.

```
uint32_t tc_get_status(
    struct tc_module *const module_inst)
```

Retrieves the status of the module, giving overall state information.

Table 25-35. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TC software instance struct

Returns

Bitmask of `TC_STATUS_*` flags.

Table 25-36. Return Values

Return value	Description
TC_STATUS_CHANNEL_0_MATCH	Timer channel 0 compare/capture match
TC_STATUS_CHANNEL_1_MATCH	Timer channel 1 compare/capture match
TC_STATUS_SYNC_READY	Timer read synchronization has completed
TC_STATUS_CAPTURE_OVERFLOW	Timer capture data has overflowed
TC_STATUS_COUNT_OVERFLOW	Timer count value has overflowed
TC_STATUS_CHN0_BUFFER_VALID	Timer count channel 0 compare/capture buffer valid
TC_STATUS_CHN1_BUFFER_VALID	Timer count channel 1 compare/capture buffer valid
TC_STATUS_PERIOD_BUFFER_VALID	Timer count period buffer valid

Function `tc_clear_status()`

Clears a module status flag.

```
void tc_clear_status(
    struct tc_module *const module_inst,
    const uint32_t status_flags)
```

Clears the given status flag of the module.

Table 25-37. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TC software instance struct
[in]	status_flags	Bitmask of TC_STATUS_* flags to clear

25.6.5. Enumeration Definitions

25.6.5.1. Waveform Inversion Mode

Enum tc_waveform_invert_output

Output waveform inversion mode.

Table 25-38. Members

Enum value	Description
TC_WAVEFORM_INVERT_OUTPUT_NONE	No inversion of the waveform output
TC_WAVEFORM_INVERT_OUTPUT_CHANNEL_0	Invert output from compare channel 0
TC_WAVEFORM_INVERT_OUTPUT_CHANNEL_1	Invert output from compare channel 1

Enum tc_event_action

Event action to perform when the module is triggered by an event.

Table 25-39. Members

Enum value	Description
TC_EVENT_ACTION_OFF	No event action
TC_EVENT_ACTION_RETRIGGER	Re-trigger on event
TC_EVENT_ACTION_INCREMENT_COUNTER	Increment counter on event
TC_EVENT_ACTION_START	Start counter on event
TC_EVENT_ACTION_PPW	Store period in capture register 0, pulse width in capture register 1
TC_EVENT_ACTION_PWP	Store pulse width in capture register 0, period in capture register 1
TC_EVENT_ACTION_STAMP	Time stamp capture
TC_EVENT_ACTION_PW	Pulse width capture

25.6.5.2. Enum tc_callback

Enum for the possible callback types for the TC module.

Table 25-40. Members

Enum value	Description
TC_CALLBACK_OVERFLOW	Callback for TC overflow
TC_CALLBACK_ERROR	Callback for capture overflow error
TC_CALLBACK_CC_CHANNEL0	Callback for capture compare channel 0
TC_CALLBACK_CC_CHANNEL1	Callback for capture compare channel 1

25.6.5.3. Enum tc_clock_prescaler

This enum is used to choose the clock prescaler configuration. The prescaler divides the clock frequency of the TC module to make the counter count slower.

Table 25-41. Members

Enum value	Description
TC_CLOCK_PRESCALER_DIV1	Divide clock by 1
TC_CLOCK_PRESCALER_DIV2	Divide clock by 2
TC_CLOCK_PRESCALER_DIV4	Divide clock by 4
TC_CLOCK_PRESCALER_DIV8	Divide clock by 8
TC_CLOCK_PRESCALER_DIV16	Divide clock by 16
TC_CLOCK_PRESCALER_DIV64	Divide clock by 64
TC_CLOCK_PRESCALER_DIV256	Divide clock by 256
TC_CLOCK_PRESCALER_DIV1024	Divide clock by 1024

25.6.5.4. Enum tc_compare_capture_channel

This enum is used to specify which capture/compare channel to do operations on.

Table 25-42. Members

Enum value	Description
TC_COMPARE_CAPTURE_CHANNEL_0	Index of compare capture channel 0
TC_COMPARE_CAPTURE_CHANNEL_1	Index of compare capture channel 1

25.6.5.5. Enum tc_count_direction

Timer/Counter count direction.

Table 25-43. Members

Enum value	Description
TC_COUNT_DIRECTION_UP	Timer should count upward from zero to MAX
TC_COUNT_DIRECTION_DOWN	Timer should count downward to zero from MAX

25.6.5.6. Enum tc_counter_size

This enum specifies the maximum value it is possible to count to.

Table 25-44. Members

Enum value	Description
TC_COUNTER_SIZE_8BIT	The counter's maximum value is 0xFF, the period register is available to be used as top value
TC_COUNTER_SIZE_16BIT	The counter's maximum value is 0xFFFF. There is no separate period register, to modify top one of the capture compare registers has to be used. This limits the amount of available channels.
TC_COUNTER_SIZE_32BIT	The counter's maximum value is 0xFFFFFFFF. There is no separate period register, to modify top one of the capture compare registers has to be used. This limits the amount of available channels.

25.6.5.7. Enum tc_reload_action

This enum specify how the counter and prescaler should reload.

Table 25-45. Members

Enum value	Description
TC_RELOAD_ACTION_GCLK	The counter is reloaded/reset on the next GCLK and starts counting on the prescaler clock
TC_RELOAD_ACTION_PRESC	The counter is reloaded/reset on the next prescaler clock
TC_RELOAD_ACTION_RESYNC	The counter is reloaded/reset on the next GCLK, and the prescaler is restarted as well

25.6.5.8. Enum tc_wave_generation

This enum is used to select which mode to run the wave generation in.

Table 25-46. Members

Enum value	Description
TC_WAVE_GENERATION_NORMAL_FREQ	Top is maximum, except in 8-bit counter size where it is the PER register
TC_WAVE_GENERATION_MATCH_FREQ	Top is CC0, except in 8-bit counter size where it is the PER register

Enum value	Description
TC_WAVE_GENERATION_NORMAL_PWM	Top is maximum, except in 8-bit counter size where it is the PER register
TC_WAVE_GENERATION_MATCH_PWM	Top is CC0, except in 8-bit counter size where it is the PER register

25.7. Extra Information for TC Driver

25.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
DMA	Direct Memory Access
TC	Timer Counter
PWM	Pulse Width Modulation
PWP	Pulse Width Period
PPW	Period Pulse Width

25.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

25.7.3. Errata

There are no errata related to this driver.

25.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added support for SAM D21 and do some modifications as below: • Clean up in the configuration structure, the counter size setting specific registers is accessed through the counter_8_bit, counter_16_bit, and counter_32_bit structures • All event related settings moved into the tc_event structure
Added automatic digital clock interface enable for the slave TC module when a timer is initialized in 32-bit mode
Initial release

25.8. Examples for TC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Timer/Counter \(TC\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that QSGs can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for TC - Basic](#)
- [Quick Start Guide for TC - Match Frequency Wave Generation](#)
- [Quick Start Guide for TC - Timer](#)
- [Quick Start Guide for TC - Callback](#)
- [Quick Start Guide for Using DMA with TC](#)

25.8.1. Quick Start Guide for TC - Basic

In this use case, the TC will be used to generate a PWM signal. Here the pulse width is set to one quarter of the period. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit resolution on the counter
- No prescaler
- Normal PWM wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Capture compare channel 0 set to $0xFFFF/4$

25.8.1.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D20 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE
```

```
#define PWM_OUT_PIN      EXT1_PWM_0_PIN
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

- **SAM R21 Xplained Pro.**

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN      EXT1_PWM_0_PIN
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

- **SAM D11 Xplained Pro.**

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN      EXT1_PWM_0_PIN
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

- **SAM L21 Xplained Pro.**

```
#define PWM_MODULE      EXT2_PWM_MODULE
#define PWM_OUT_PIN      EXT2_PWM_0_PIN
#define PWM_OUT_MUX      EXT2_PWM_0_MUX
```

- **SAM L22 Xplained Pro.**

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN      EXT1_PWM_0_PIN
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

- **SAM DA1 Xplained Pro.**

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN      EXT1_PWM_0_PIN
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

- **SAM C21 Xplained Pro.**

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN      EXT1_PWM_0_PIN
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

Add to the main application source file, outside of any functions:

```
struct tc_module tc_instance;
```

Copy-paste the following setup code to your user application:

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation   = TC_WAVE_GENERATION_NORMAL_PWM;
    config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF / 4);
}
```

```

config_tc.pwm_channel[0].enabled = true;
config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

tc_init(&tc_instance, PWM_MODULE, &config_tc);

tc_enable(&tc_instance);
}

```

Add to user application initialization (typically the start of `main()`):

```
configure_tc();
```

Workflow

1. Create a module software instance structure for the TC module to store the TC driver state while it is in use.

```
struct tc_module tc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TC module.
 1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

```
struct tc_config config_tc;
```

2. Initialize the TC configuration struct with the module's default values.

```
tc_get_config_defaults(&config_tc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```

config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;
config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF /
4);

```

4. Alter the TC settings to configure the PWM output on a physical device pin.

```

config_tc.pwm_channel[0].enabled = true;
config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

```

5. Configure the TC module with the desired settings.

```
tc_init(&tc_instance, PWM_MODULE, &config_tc);
```

6. Enable the TC module to start the timer and begin PWM signal generation.

```
tc_enable(&tc_instance);
```


25.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    /* Infinite loop */  
}
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TC module.

```
while (true) {  
    /* Infinite loop */  
}
```

25.8.2. Quick Start Guide for TC - Match Frequency Wave Generation

In this use case, the TC will be used to generate a match frequency. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit resolution on the counter
- No prescaler
- Match frequency wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Capture compare channel 0 set to 4000

When system clock is 8MHz, and the compare channel 0 is 4000, the output frequency will be about 1KHz ($8000000/4000/2$).

25.8.2.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE  
  
#define PWM_OUT_PIN     EXT1_PWM_0_PIN  
  
#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D20 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

Add to the main application source file, outside of any functions:

```
struct tc_module tc_instance;
```

Copy-paste the following setup code to your user application:

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation   = TC_WAVE_GENERATION_MATCH_FREQ;
    config_tc.counter_16_bit.compare_capture_channel[0] = 4000;

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);

    tc_enable(&tc_instance);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_tc();
```

Workflow

1. Create a module software instance structure for the TC module to store the TC driver state while it is in use.

```
struct tc_module tc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TC module.
 1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

```
struct tc_config config_tc;
```

2. Initialize the TC configuration struct with the module's default values.

```
tc_get_config_defaults(&config_tc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```
config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;  
config_tc.wave_generation = TC_WAVE_GENERATION_MATCH_FREQ;  
config_tc.counter_16_bit.compare_capture_channel[0] = 4000;
```

4. Alter the TC settings to configure the match frequency output on a physical device pin.

```
config_tc.pwm_channel[0].enabled = true;  
config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;  
config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
```

5. Configure the TC module with the desired settings.

```
tc_init(&tc_instance, PWM_MODULE, &config_tc);
```

6. Enable the TC module to start the timer and begin match frequency wave generation.

```
tc_enable(&tc_instance);
```

25.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    /* Infinite loop */  
}
```

Workflow

1. Enter an infinite loop while the match frequency wave is generated via the TC module.

```
while (true) {  
    /* Infinite loop */  
}
```

25.8.3. Quick Start Guide for TC - Timer

In this use case, the TC will be used as a timer to generate overflow and compare match callbacks. In the callbacks the on-board LED is toggled.

The TC module will be set up as follows:

- GCLK generator 1 (GCLK 32K) clock source
- 16-bit resolution on the counter
- Prescaler is divided by 64
- GCLK reload action
- Count upward
- Don't run in standby
- No waveform outputs
- No capture enabled
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0

- Counter top set to 2000 (about 4s) and generate overflow callback
- Channel 0 is set to compare and match value 900 and generate callback
- Channel 1 is set to compare and match value 930 and generate callback

25.8.3.1. Quick Start

Prerequisites

For this use case, XOSC32K should be enabled and available through GCLK generator 1 clock source selection. Within Atmel Software Framework (ASF) it can be done through modifying *conf_clocks.h*. See System Clock Management Driver for more details about clock configuration.

Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D20 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

- SAM D21 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

- SAM R21 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

- SAM D11 Xplained Pro.

```
#define CONF_TC_MODULE TC1
```

- SAM L21 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

- SAM L22 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

- SAM DA1 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

- SAM C21 Xplained Pro.

```
#define CONF_TC_MODULE TC3
```

Add to the main application source file, outside of any functions:

```
struct tc_module tc_instance;
```

Copy-paste the following callback function code to your user application:

```
void tc_callback_to_toggle_led(
    struct tc_module *const module_inst)
{
    port_pin_toggle_output_level(LED0_PIN);
}
```

Copy-paste the following setup code to your user application:

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);
```

```

    config_tc.counter_size = TC_COUNTER_SIZE_8BIT;
    config_tc.clock_source = GCLK_GENERATOR_1;
    config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV1024;
    config_tc.counter_8_bit.period = 100;
    config_tc.counter_8_bit.compare_capture_channel[0] = 50;
    config_tc.counter_8_bit.compare_capture_channel[1] = 54;

    tc_init(&tc_instance, CONF_TC_MODULE, &config_tc);

    tc_enable(&tc_instance);
}

void configure_tc_callbacks(void)
{
    tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
        TC_CALLBACK_OVERFLOW);
    tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
        TC_CALLBACK_CC_CHANNEL0);
    tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
        TC_CALLBACK_CC_CHANNEL1);

    tc_enable_callback(&tc_instance, TC_CALLBACK_OVERFLOW);
    tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
    tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL1);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_tc();
configure_tc_callbacks();

```

Workflow

1. Create a module software instance structure for the TC module to store the TC driver state while it is in use.

```

struct tc_module tc_instance;

```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TC module.
 1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

```

struct tc_config config_tc;

```

2. Initialize the TC configuration struct with the module's default values.

```

tc_get_config_defaults(&config_tc);

```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TC settings to configure the GCLK source, prescaler, period, and compare channel values.

```

config_tc.counter_size = TC_COUNTER_SIZE_8BIT;
config_tc.clock_source = GCLK_GENERATOR_1;
config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV1024;
config_tc.counter_8_bit.period = 100;
config_tc.counter_8_bit.compare_capture_channel[0] = 50;
config_tc.counter_8_bit.compare_capture_channel[1] = 54;

```

4. Configure the TC module with the desired settings.

```
tc_init(&tc_instance, CONF_TC_MODULE, &config_tc);
```

5. Enable the TC module to start the timer.

```
tc_enable(&tc_instance);
```

3. Configure the TC callbacks.

1. Register the Overflow and Compare Channel Match callback functions with the driver.

```
tc_register_callback(&tc_instance, tc_callback_to_toggle_led,  
    TC_CALLBACK_OVERFLOW);  
tc_register_callback(&tc_instance, tc_callback_to_toggle_led,  
    TC_CALLBACK_CC_CHANNEL0);  
tc_register_callback(&tc_instance, tc_callback_to_toggle_led,  
    TC_CALLBACK_CC_CHANNEL1);
```

2. Enable the Overflow and Compare Channel Match callbacks so that it will be called by the driver when appropriate.

```
tc_enable_callback(&tc_instance, TC_CALLBACK_OVERFLOW);  
tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);  
tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL1);
```

25.8.3.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();  
  
while (true) {  
}
```

Workflow

1. Enter an infinite loop while the timer is running.

```
while (true) {  
}
```

25.8.4. Quick Start Guide for TC - Callback

In this use case, the TC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit resolution on the counter
- No prescaler
- Normal PWM wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations

- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0

25.8.4.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN     EXT1_PWM_0_PIN
#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D20 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN     EXT1_PWM_0_PIN
#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM R21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN     EXT1_PWM_0_PIN
#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D11 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE
#define PWM_OUT_PIN     EXT1_PWM_0_PIN
#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM L21 Xplained Pro.

```
#define PWM_MODULE      EXT2_PWM_MODULE
#define PWM_OUT_PIN     EXT2_PWM_0_PIN
#define PWM_OUT_MUX     EXT2_PWM_0_MUX
```

- SAM L22 Xplained Pro.

```
#define PWM_MODULE      EXT3_PWM_MODULE
#define PWM_OUT_PIN     EXT3_PWM_0_PIN
#define PWM_OUT_MUX     EXT3_PWM_0_MUX
```

- SAM DA1 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM C21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

Add to the main application source file, outside of any functions:

```
struct tc_module tc_instance;
```

Copy-paste the following callback function code to your user application:

```
void tc_callback_to_change_duty_cycle(
    struct tc_module *const module_inst)
{
    static uint16_t i = 0;

    i += 128;
    tc_set_compare_value(module_inst, TC_COMPARE_CAPTURE_CHANNEL_0, i
+ 1);
}
```

Copy-paste the following setup code to your user application:

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation   = TC_WAVE_GENERATION_NORMAL_PWM;
    config_tc.counter_16_bit.compare_capture_channel[0] = 0xFFFF;

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);
    tc_enable(&tc_instance);
}

void configure_tc_callbacks(void)
{
    tc_register_callback(
        &tc_instance,
        tc_callback_to_change_duty_cycle,
        TC_CALLBACK_CC_CHANNEL0);

    tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
}
```


Add to user application initialization (typically the start of `main()`):

```
configure_tc();  
configure_tc_callbacks();
```

Workflow

1. Create a module software instance structure for the TC module to store the TC driver state while it is in use.

```
struct tc_module tc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TC module.
 1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

```
struct tc_config config_tc;
```

2. Initialize the TC configuration struct with the module's default values.

```
tc_get_config_defaults(&config_tc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```
config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;  
config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;  
config_tc.counter_16_bit.compare_capture_channel[0] = 0xFFFF;
```

4. Alter the TC settings to configure the PWM output on a physical device pin.

```
config_tc.pwm_channel[0].enabled = true;  
config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;  
config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
```

5. Configure the TC module with the desired settings.

```
tc_init(&tc_instance, PWM_MODULE, &config_tc);
```

6. Enable the TC module to start the timer and begin PWM signal generation.

```
tc_enable(&tc_instance);
```

3. Configure the TC callbacks.

1. Register the Compare Channel 0 Match callback functions with the driver.

```
tc_register_callback(  
    &tc_instance,  
    tc_callback_to_change_duty_cycle,  
    TC_CALLBACK_CC_CHANNEL0);
```

2. Enable the Compare Channel 0 Match callback so that it will be called by the driver when appropriate.

```
tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
```

25.8.4.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();  
  
while (true) {  
}
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TC module.

```
while (true) {  
}
```

25.8.5. Quick Start Guide for Using DMA with TC

The supported kit list:

- SAM D21/R21/D11/L21/L22/DA1/C21 Xplained Pro

In this use case, the TC will be used to generate a PWM signal. Here the pulse width is set to one quarter of the period. Once the counter value matches the values in the Compare/Capture Value register, an event will be triggered for a DMA memory to memory transfer. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit resolution on the counter
- No prescaler
- Normal PWM wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Capture compare channel 0 set to 0xFFFF/4

The DMA module is configured for:

- Move data from memory to memory
- Using peripheral trigger of TC6 Match/Compare 0
- Using DMA priority level 0

25.8.5.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions, according to the kit used:

- **SAM D21 Xplained Pro.**

```
#define PWM_MODULE          EXT1_PWM_MODULE

#define PWM_OUT_PIN         EXT1_PWM_0_PIN

#define PWM_OUT_MUX         EXT1_PWM_0_MUX

#define M2M_DMAC_TRIGGER_ID TC6_DMAC_ID_MC_0
```

- **SAM R21 Xplained Pro.**

```
#define PWM_MODULE          EXT1_PWM_MODULE

#define PWM_OUT_PIN         EXT1_PWM_0_PIN

#define PWM_OUT_MUX         EXT1_PWM_0_MUX

#define M2M_DMAC_TRIGGER_ID TC3_DMAC_ID_MC_0
```

- **SAM D11 Xplained Pro.**

```
#define PWM_MODULE          EXT1_PWM_MODULE

#define PWM_OUT_PIN         EXT1_PWM_0_PIN

#define PWM_OUT_MUX         EXT1_PWM_0_MUX

#define M2M_DMAC_TRIGGER_ID TC1_DMAC_ID_MC_0
```

- **SAM L21 Xplained Pro.**

```
#define PWM_MODULE          EXT2_PWM_MODULE

#define PWM_OUT_PIN         EXT2_PWM_0_PIN

#define PWM_OUT_MUX         EXT2_PWM_0_MUX

#define M2M_DMAC_TRIGGER_ID TC0_DMAC_ID_MC_0
```

- **SAM L22 Xplained Pro.**

```
#define PWM_MODULE          EXT1_PWM_MODULE

#define PWM_OUT_PIN         EXT1_PWM_0_PIN

#define PWM_OUT_MUX         EXT1_PWM_0_MUX

#define M2M_DMAC_TRIGGER_ID TC0_DMAC_ID_MC_0
```

- **SAM DA1 Xplained Pro.**

```
#define PWM_MODULE          EXT1_PWM_MODULE

#define PWM_OUT_PIN         EXT1_PWM_0_PIN

#define PWM_OUT_MUX         EXT1_PWM_0_MUX

#define M2M_DMAC_TRIGGER_ID TC6_DMAC_ID_MC_0
```

- SAM C21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX

#define M2M_DMACE_TRIGGER_ID TC0_DMACE_ID_MC_0
```

Add to the main application source file, outside of any functions:

```
struct tc_module tc_instance;

struct dma_resource example_resource;

#define TRANSFER_SIZE      (16)

#define TRANSFER_COUNTER   (32)

static uint8_t source_memory[TRANSFER_SIZE*TRANSFER_COUNTER];

static uint8_t destination_memory[TRANSFER_SIZE*TRANSFER_COUNTER];

static volatile bool transfer_is_done = false;

COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor SECTION_DMACE_DESCRIPTOR;
```

Copy-paste the following setup code to your user application:

```
#define TRANSFER_SIZE      (16)

#define TRANSFER_COUNTER   (32)

static uint8_t source_memory[TRANSFER_SIZE*TRANSFER_COUNTER];

static uint8_t destination_memory[TRANSFER_SIZE*TRANSFER_COUNTER];

static volatile bool transfer_is_done = false;

COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor SECTION_DMACE_DESCRIPTOR;

void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation   = TC_WAVE_GENERATION_NORMAL_PWM;
    config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF / 4);

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);

    tc_enable(&tc_instance);
}
```

```

}

void transfer_done(struct dma_resource* const resource )
{
    UNUSED(resource);

    transfer_is_done = true;
}

void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);
    config.peripheral_trigger = M2M_DMAC_TRIGGER_ID;

    dma_allocate(resource, &config);
}

void setup_dma_descriptor(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = TRANSFER_SIZE;
    descriptor_config.source_address = (uint32_t)source_memory +
TRANSFER_SIZE;
    descriptor_config.destination_address =
        (uint32_t)destination_memory + TRANSFER_SIZE;

    dma_descriptor_create(descriptor, &descriptor_config);
}

```

Add to user application initialization (typically the start of `main()`):

```
configure_tc();
```

Workflow

Create variables

1. Create a module software instance structure for the TC module to store the TC driver state while it is in use.

```
struct tc_module tc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a module software instance structure for DMA resource to store the DMA resource state while it is in use.

```
struct dma_resource example_resource;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

Configure TC

1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

```
struct tc_config config_tc;
```

2. Initialize the TC configuration struct with the module's default values.

```
tc_get_config_defaults(&config_tc);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```
config_tc.counter_size      = TC_COUNTER_SIZE_16BIT;
config_tc.wave_generation   = TC_WAVE_GENERATION_NORMAL_PWM;
config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF / 4);
```

4. Alter the TC settings to configure the PWM output on a physical device pin.

```
config_tc.pwm_channel[0].enabled = true;
config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
```

5. Configure the TC module with the desired settings.

```
tc_init(&tc_instance, PWM_MODULE, &config_tc);
```

6. Enable the TC module to start the timer and begin PWM signal generation.

```
tc_enable(&tc_instance);
```

Configure DMA

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config config;
```

2. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&config);
config.peripheral_trigger = M2M_DMAC_TRIGGER_ID;
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

4. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

5. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

6. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
descriptor_config.block_transfer_count = TRANSFER_SIZE;
descriptor_config.source_address = (uint32_t)source_memory +
TRANSFER_SIZE;
```

```
descriptor_config.destination_address =
    (uint32_t)destination_memory + TRANSFER_SIZE;
```

7. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

8. Add the DMA transfer descriptor to the allocated DMA resource.

```
dma_add_descriptor(&example_resource, &example_descriptor);
```

9. Register a callback to indicate transfer status.

```
dma_register_callback(&example_resource, transfer_done,
    DMA_CALLBACK_TRANSFER_DONE);
```

10. The transfer done flag is set in the registered callback function.

```
void transfer_done(struct dma_resource* const resource )
{
    UNUSED(resource);

    transfer_is_done = true;
}
```

Prepare data

1. Setup memory content for validate transfer.

```
for (i = 0; i < TRANSFER_SIZE*TRANSFER_COUNTER; i++) {
    source_memory[i] = i;
}
```

25.8.5.2. Use Case

Code

Copy-paste the following code to your user application:

```
for(i=0;i<TRANSFER_COUNTER;i++) {
    transfer_is_done = false;

    dma_start_transfer_job(&example_resource);

    while (!transfer_is_done) {
        /* Wait for transfer done */
    }

    example_descriptor.SRCADDR.reg += TRANSFER_SIZE;
    example_descriptor.DSTADDR.reg += TRANSFER_SIZE;
}

while(1);
```

Workflow

1. Start the loop for transfer.

```
for(i=0;i<TRANSFER_COUNTER;i++) {
    transfer_is_done = false;

    dma_start_transfer_job(&example_resource);

    while (!transfer_is_done) {
        /* Wait for transfer done */
    }
}
```

```
example_descriptor.SRCADDR.reg += TRANSFER_SIZE;  
example_descriptor.DSTADDR.reg += TRANSFER_SIZE;  
}
```

2. Set the transfer done flag as false.

```
transfer_is_done = false;
```

3. Start the transfer job.

```
dma_start_transfer_job(&example_resource);
```

4. Wait for transfer done.

```
while (!transfer_is_done) {  
    /* Wait for transfer done */  
}
```

5. Update the source and destination address for next transfer.

```
example_descriptor.SRCADDR.reg += TRANSFER_SIZE;  
example_descriptor.DSTADDR.reg += TRANSFER_SIZE;
```

6. Enter endless loop.

```
while(1);
```


26. SAM Timer Counter for Control Applications (TCC) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the TCC module within the device, for waveform generation and timing operations. It also provides extended options for control applications.

The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- TCC (Timer/Counter for Control Applications)

The following devices can use this module:

- Atmel | SMART SAM D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

26.1. Prerequisites

There are no prerequisites for this module.

26.2. Module Overview

The Timer/Counter for Control Applications (TCC) module provides a set of timing and counting related functionality, such as the generation of periodic waveforms, the capturing of a periodic waveform's frequency/duty cycle, software timekeeping for periodic operations, waveform extension control, fault detection etc.

The counter size of the TCC modules can be 16- or 24-bit depending on the TCC instance. Refer [SAM TCC Feature List](#) and [SAM D10/D11 TCC Feature List](#) for details on TCC instances.

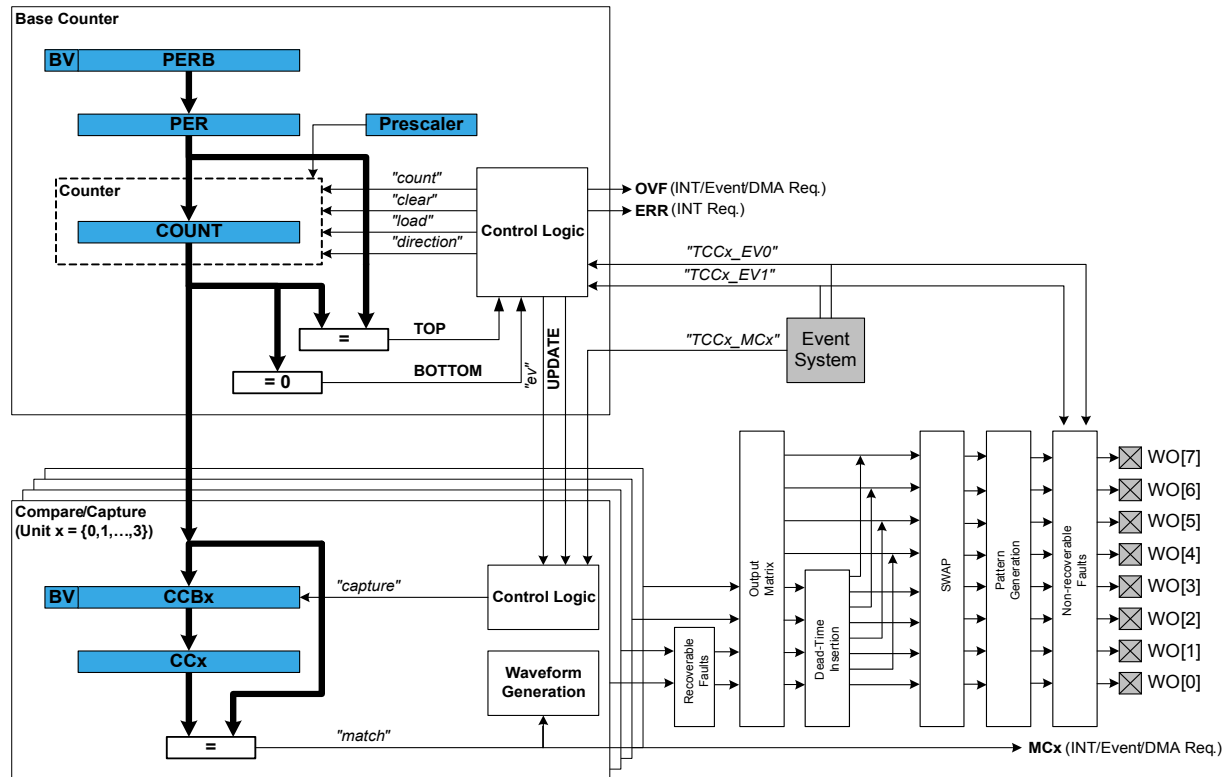
The TCC module for the SAM includes the following functions:

- Generation of PWM signals
- Generation of timestamps for events
- General time counting
- Waveform period capture

- Waveform frequency capture
- Additional control for generated waveform outputs
- Fault protection for waveform generation

Figure 26-1 shows the overview of the TCC Module.

Figure 26-1. Overview of the TCC Module



26.2.1. Functional Description

The TCC module consists of following sections:

- Base Counter
- Compare/Capture channels, with waveform generation
- Waveform extension control and fault detection
- Interface to the event system, DMAC, and the interrupt system

The base counter can be configured to either count a prescaled generic clock or events from the event system. (TCCx, with event action configured to counting). The counter value can be used by compare/capture channels which can be set up either in compare mode or capture mode.

In capture mode, the counter value is stored when a configurable event occurs. This mode can be used to generate timestamps used in event capture, or it can be used for the measurement of a periodic input signal's frequency/duty cycle.

In compare mode, the counter value is compared against one or more of the configured channels' compare values. When the counter value coincides with a compare value an action can be taken automatically by the module, such as generating an output event or toggling a pin when used for frequency or PWM signal generation.

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the the input event of another. For more information on event routing, refer to the event driver documentation.

In compare mode, when output signal is generated, extended waveform controls are available, to arrange the compare outputs into specific formats. The Output matrix can change the channel output routing. Pattern generation unit can overwrite the output signal line to specific state. The Fault protection feature of the TCC supports recoverable and non-recoverable faults.

26.2.2. Base Timer/Counter

26.2.2.1. Timer/Counter Size

Each TCC has a counter size of either 16- or 24-bits. The size of the counter determines the maximum value it can count to before an overflow occurs. [Table 26-1](#) shows the maximum values for each of the possible counter sizes.

Table 26-1. Timer Counter Sizes and Their Maximum Count Values

Counter size	Max. (hexadecimal)	Max. (decimal)
16-bit	0xFFFF	65,535
24-bit	0xFFFFFF	16,777,215

The period/top value of the counter can be set, to define counting period. This will allow the counter to overflow when the counter value reaches the period/top value.

26.2.2.2. Timer/Counter Clock and Prescaler

TCC is clocked asynchronously to the system clock by a GCLK (Generic Clock) channel. The GCLK channel can be connected to any of the GCLK generators. The GCLK generators are configured to use one of the available clock sources in the system such as internal oscillator, external crystals, etc. See the Generic Clock driver for more information.

Each TCC module in the SAM has its own individual clock prescaler, which can be used to divide the input clock frequency used by the counter. This prescaler only scales the clock used to provide clock pulses for the counter to count, and does not affect the digital register interface portion of the module, thus the timer registers will be synchronized to the raw GCLK frequency input to the module.

As a result of this, when selecting a GCLK frequency and timer prescaler value, the user application should consider both the timer resolution required and the synchronization frequency to avoid lengthy synchronization times of the module if a very slow GCLK frequency is fed into the TCC module. It is preferable to use a higher module GCLK frequency as the input to the timer, and prescale this down as much as possible to obtain a suitable counter frequency in latency-sensitive applications.

26.2.2.3. Timer/Counter Control Inputs (Events)

The TCC can take several actions on the occurrence of an input event. The event actions are listed in [Table 26-2](#).

Table 26-2. TCC Module Event Actions

Event action	Description	Applied event
TCC_EVENT_ACTION_OFF	No action on the event input	All
TCC_EVENT_ACTION_RETRIGGER	Re-trigger Counter on event	All
TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event	All
TCC_EVENT_ACTION_START	Counter start on event	EV0
TCC_EVENT_ACTION_DIR_CONTROL	Counter direction control	EV0
TCC_EVENT_ACTION_DECREMENT	Counter decrement on event	EV0
TCC_EVENT_ACTION_PERIOD_PULSE_WIDTH_CAPTURE	Capture pulse period and pulse width	EV0
TCC_EVENT_ACTION_PULSE_WIDTH_PERIOD_CAPTURE	Capture pulse width and pulse period	EV0
TCC_EVENT_ACTION_STOP	Counter stop on event	EV1
TCC_EVENT_ACTION_COUNT_EVENT	Counter count on event	EV1
TCC_EVENT_ACTION_INCREMENT	Counter increment on event	EV1
TCC_EVENT_ACTION_COUNT_DURING_ACTIVE	Counter count during active state of asynchronous event	EV1

26.2.2.4. Timer/Counter Reloading

The TCC also has a configurable reload action, used when a re-trigger event occurs. Examples of a re-trigger event could be the counter reaching the maximum value when counting up, or when an event from the event system makes the counter to re-trigger. The reload action determines if the prescaler should be reset, and on which clock. The counter will always be reloaded with the value it is set to start counting. The user can choose between three different reload actions, described in [Table 26-3](#).

Table 26-3. TCC Module Reload Actions

Reload action	Description
TCC_RELOAD_ACTION_GCLK	Reload TCC counter value on next GCLK cycle. Leave prescaler as-is.
TCC_RELOAD_ACTION_PRESC	Reloads TCC counter value on next prescaler clock. Leave prescaler as-is.
TCC_RELOAD_ACTION_RESYNC	Reload TCC counter value on next GCLK cycle. Clear prescaler to zero.

The reload action to use will depend on the specific application being implemented. One example is when an external trigger for a reload occurs; if the TCC uses the prescaler, the counter in the prescaler should not have a value between zero and the division factor. The counter in the TCC module and the counter in the prescaler should both start at zero. If the counter is set to re-trigger when it reaches the maximum value, this is not the right option to use. In such a case it would be better if the prescaler is left unaltered when the re-trigger happens, letting the counter reset on the next GCLK cycle.

26.2.2.5. One-shot Mode

The TCC module can be configured in one-shot mode. When configured in this manner, starting the timer will cause it to count until the next overflow or underflow condition before automatically halting, waiting to be manually triggered by the user application software or an event from the event system.

26.2.3. Capture Operations

In capture operations, any event from the event system or a pin change can trigger a capture of the counter value. This captured counter value can be used as timestamps for the events, or it can be used in frequency and pulse width capture.

26.2.3.1. Capture Operations - Event

Event capture is a simple use of the capture functionality, designed to create timestamps for specific events. When the input event appears, the current counter value is copied into the corresponding compare/capture register, which can then be read by the user application.

Note that when performing any capture operation, there is a risk that the counter reaches its top value (MAX) when counting up, or the bottom value (zero) when counting down, before the capture event occurs. This can distort the result, making event timestamps to appear shorter than they really are. In this case, the user application should check for timer overflow when reading a capture result in order to detect this situation and perform an appropriate adjustment.

Before checking for a new capture, [TCC_STATUS_COUNT_OVERFLOW](#) should be checked. The response to an overflow error is left to the user application, however, it may be necessary to clear both the overflow flag and the capture flag upon each capture reading.

26.2.3.2. Capture Operations - Pulse Width

Pulse Width Capture mode makes it possible to measure the pulse width and period of PWM signals. This mode uses two capture channels of the counter. There are two modes for pulse width capture; Pulse Width Period (PWP) and Period Pulse Width (PPW). In PWP mode, capture channel 0 is used for storing the pulse width and capture channel 1 stores the observed period. While in PPW mode, the roles of the two capture channels are reversed.

As in the above example it is necessary to poll on interrupt flags to see if a new capture has happened and check that a capture overflow error has not occurred.

Refer to [Timer/Counter Control Inputs \(Events\)](#) to set up the input event to perform pulse width capture.

26.2.4. Compare Match Operation

In compare match operation, Compare/Capture registers are compared with the counter value. When the timer's count value matches the value of a compare channel, a user defined action can be taken.

26.2.4.1. Basic Timer

A Basic Timer is a simple application where compare match operation is used to determine when a specific period has elapsed. In Basic Timer operations, one or more values in the module's Compare/Capture registers are used to specify the time (in terms of the number of prescaled GCLK cycles, or input events) at which an action should be taken by the microcontroller. This can be an Interrupt Service Routine (ISR), event generation via the event system, or a software flag that is polled from the user application.

26.2.4.2. Waveform Generation

Waveform generation enables the TCC module to generate square waves, or, if combined with an external passive low-pass filter, analog waveforms.

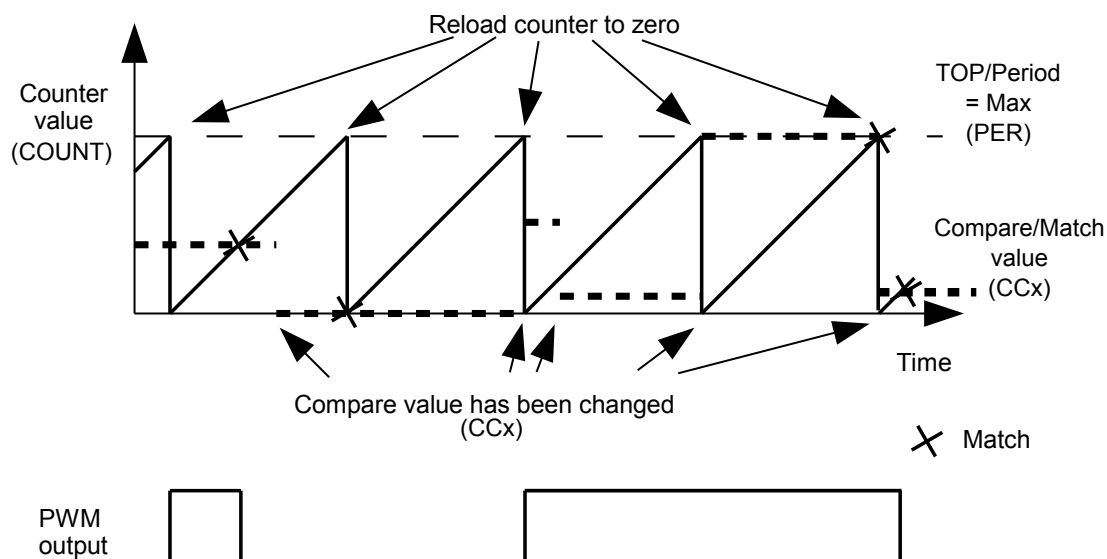
26.2.4.3. Waveform Generation - PWM

Pulse width modulation is a form of waveform generation and a signalling technique that can be useful in many applications. When PWM mode is used, a digital pulse train with a configurable frequency and duty cycle can be generated by the TCC module and output to a GPIO pin of the device.

Often PWM is used to communicate a control or information parameter to an external circuit or component. Differing impedances of the source generator and sink receiver circuits is less of an issue when using PWM compared to using an analog voltage value, as noise will not generally affect the signal's integrity to a meaningful extent.

[Figure 26-2](#) illustrates operations and different states of the counter and its output when using the timer in Normal PWM mode (Single Slope). As can be seen, the TOP/PERIOD value is unchanged and is set to MAX. The compare match value is changed at several points to illustrate the resulting waveform output changes. The PWM output is set to normal (i.e. non-inverted) output mode.

Figure 26-2. Example Of PWM In Single-Slope Mode, and Different Counter Operations



Several PWM modes are supported by the TCC module, refer to datasheet for the details on PWM waveform generation.

26.2.4.4. Waveform Generation - Frequency

Normal Frequency Generation is in many ways identical to PWM generation. However, only in Frequency Generation, a toggle occurs on the output when a match on a compare channels occurs.

When the Match Frequency Generation is used, the timer value is reset on match condition, resulting in a variable frequency square wave with a fixed 50% duty cycle.

26.2.5. Waveform Extended Controls

26.2.5.1. Pattern Generation

Pattern insertion allows the TCC module to change the actual pin output level without modifying the compare/match settings.

Table 26-4. TCC Module Output Pattern Generation

Pattern	Description
TCC_OUTPUT_PATTERN_DISABLE	Pattern disabled, generate output as is
TCC_OUTPUT_PATTERN_0	Generate pattern 0 on output (keep the output LOW)
TCC_OUTPUT_PATTERN_1	Generate pattern 1 on output (keep the output HIGH)

26.2.5.2. Recoverable Faults

The recoverable faults can trigger one or several of following fault actions:

1. ***Halt*** action: The recoverable faults can halt the TCC timer/counter, so that the final output wave is kept at a defined state. When the fault state is removed it is possible to recover the counter and waveform generation. The halt action is defined as:

Table 26-5. TCC Module Recoverable Fault Halt Actions

Action	Description
TCC_FAULT_HALT_ACTION_DISABLE	Halt action is disabled
TCC_FAULT_HALT_ACTION_HW_HALT	The timer/counter is halted as long as the corresponding fault is present
TCC_FAULT_HALT_ACTION_SW_HALT	The timer/counter is halted until the corresponding fault is removed and fault state cleared by software
TCC_FAULT_HALT_ACTION_NON_RECOVERABLE	Force all the TCC output pins to a pre-defined level, as what Non-Recoverable Fault do

2. ***Restart*** action: When enabled, the recoverable faults can restart the TCC timer/counter.
3. ***Keep*** action: When enabled, the recoverable faults can keep the corresponding channel output to zero when the fault condition is present.
4. ***Capture*** action: When the recoverable fault occurs, the capture action can time stamps the corresponding fault. The following capture mode is supported:

Table 26-6. TCC Module Recoverable Fault Capture Actions

Action	Description
TCC_FAULT_CAPTURE_DISABLE	Capture action is disabled
TCC_FAULT_CAPTURE_EACH	Equivalent to standard capture operation, on each fault occurrence the time stamp is captured
TCC_FAULT_CAPTURE_MINIMUM	Get the minimum time stamped value in all time stamps
TCC_FAULT_CAPTURE_MAXIMUM	Get the maximum time stamped value in all time stamps
TCC_FAULT_CAPTURE_SMALLER	Time stamp the fault input if the value is smaller than last one
TCC_FAULT_CAPTURE_BIGGER	Time stamp the fault input if the value is bigger than last one
TCC_FAULT_CAPTURE_CHANGE	Time stamp the fault input if the time stamps changes its increment direction

In TCC module, only the first two compare channels (CC0 and CC1) can work with recoverable fault inputs. The corresponding event inputs (TCCx MC0 and TCCx MC1) are then used as fault inputs respectively. The faults are called Fault A and Fault B.

The recoverable fault can be filtered or effected by corresponding channel output. On fault condition there are many other settings that can be chosen. Refer to data sheet for more details about the recoverable fault operations.

26.2.5.3. Non-Recoverable Faults

The non-recoverable faults force all the TCC output pins to a pre-defined level (can be forced to 0 or 1). The input control signal of non-recoverable fault is from timer/counter event (TCCx EV0 and TCCx EV1). To enable non-recoverable fault, corresponding TCCx event action must be set to non-recoverable fault action ([TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT](#)). Refer to [Timer/Counter Control Inputs \(Events\)](#) to see the available event input action.

26.2.6. Double and Circular Buffering

The pattern, period, and the compare channels registers are double buffered. For these options there are effective registers (PATT, PER, and CCx) and buffer registers (PATTB, PERB, and CCx). When writing to the buffer registers, the values are buffered and will be committed to effective registers on UPDATE condition.

Usually the buffered value is cleared after it is committed, but there is also an option to circular the register buffers. The period (PER) and four lowest compare channels register (CCx, x is 0 ~ 3) support this function. When circular buffer is used, on UPDATE the previous period or compare values are copied back into the corresponding period buffer and compare buffers. This way, the register value and its buffer register value is actually switched on UPDATE condition, and will be switched back on next UPDATE condition.

For input capture, the buffer register (CCBx) and the corresponding capture channel register (CCx) act like a FIFO. When regular register (CCx) is empty or read, any content in the buffer register is passed to regular one.

In TCC module driver, when the double buffering write is enabled, any write through `tcc_set_top_value()`, `tcc_set_compare_value()`, and `tcc_set_pattern()` will be done to the corresponding buffer register. Then the value in the buffer register will be transferred to the regular register on the next UPDATE condition or by a force UPDATE using `tcc_force_double_buffer_update()`.

26.2.7. Sleep Mode

TCC modules can be configured to operate in any sleep mode, with its "run in standby" function enabled. It can wake up the device using interrupts or perform internal actions with the help of the Event System.

26.3. Special Considerations

26.3.1. Driver Feature Macro Definition

Table 26-7 shows some specific features of the TCC Module.

Table 26-7. TCC Module Specific Features

Driver Feature Macro	Supported devices
FEATURE_TCC_GENERATE_DMA_TRIGGER	SAM L21/L22/R30

Note: The specific features are only available in the driver when the selected device supports those features.

26.3.2. Module Features

The features of TCC, such as timer/counter size, number of compare capture channels, and number of outputs, are dependent on the TCC module instance being used.

26.3.2.1. SAM TCC Feature List

For SAM D21/R21/L21/L22/DA1/C21, the TCC features are:

Table 26-8. TCC module features for SAM D21/R21/L21/L22/DA1/C21

TCC#	Match/ Capture channels	Wave outputs	Counter size [bits]	Fault	Dithering	Output matrix	Dead-Time insertion	SWAP	Pattern
0	4	8	24	Y	Y	Y	Y	Y	Y
1	2	4	24	Y	Y				Y
2	2	2	16	Y					

26.3.2.2. SAM D10/D11 TCC Feature List

For SAM D10/D11, the TCC features are:

Table 26-9. TCC Module Features For SAM D10/D11

TCC#	Match/ Capture channels	Wave outputs	Counter size [bits]	Fault	Dithering	Output matrix	Dead-Time insertion	SWAP	Pattern
0	4	8	24	Y	Y	Y	Y	Y	Y

26.3.3. Channels vs. Pinouts

As the TCC module may have more waveform output pins than the number of compare/capture channels, the free pins (with number higher than number of channels) will reuse the waveform generated by channels subsequently. E.g., if the number of channels is four and the number of wave output pins is eight, channel 0 output will be available on out pin 0 and 4, channel 1 output on wave out pin 1 and 5, and so on.

26.4. Extra Information

For extra information, see [Extra Information for TCC Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

26.5. Examples

For a list of examples related to this driver, see [Examples for TCC Driver](#).

26.6. API Overview

26.6.1. Variable and Type Definitions

26.6.1.1. Type `tcc_callback_t`

```
typedef void(* tcc_callback_t )(struct tcc_module *const module)
```

Type definition for the TCC callback function.

26.6.2. Structure Definitions

26.6.2.1. Struct `tcc_capture_config`

Structure used when configuring TCC channels in capture mode.

Table 26-10. Members

Type	Name	Description
enum tcc_channel_function	<code>channel_function[]</code>	Channel functions selection (capture/match)

26.6.2.2. Struct `tcc_config`

Configuration struct for a TCC instance. This structure should be initialized by the [tcc_get_config_defaults](#) function before being modified by the user application.

Table 26-11. Members

Type	Name	Description
union <code>tcc_config.@1</code>	<code>@1</code>	TCC match/capture configurations
struct tcc_counter_config	<code>counter</code>	Structure for configuring TCC base timer/counter
bool	<code>double_buffering_enabled</code>	Set to <code>true</code> to enable double buffering write. When enabled any write through tcc_set_top_value() , tcc_set_compare_value() and tcc_set_pattern() will direct to the buffer register as buffered value, and the buffered value will be committed to effective register on UPDATE condition, if update is not locked. Note: The init values in tcc_config for tcc_init are always filled to effective registers, no matter if double buffering is enabled or not.
struct tcc_pins_config	<code>pins</code>	Structure for configuring TCC output pins
bool	<code>run_in_standby</code>	When <code>true</code> the module is enabled during standby
struct tcc_wave_extension_config	<code>wave_ext</code>	Structure for configuring TCC waveform extension

26.6.2.3. Union `tcc_config.__unnamed__`

TCC match/capture configurations

Table 26-12. Members

Type	Name	Description
struct tcc_capture_config	capture	Helps to configure a TCC channel in capture mode
struct tcc_match_wave_config	compare	For configuring a TCC channel in compare mode
struct tcc_match_wave_config	wave	Serves the same purpose as compare. Used as an alias for compare, when a TCC channel is configured for wave generation

26.6.2.4. Struct [tcc_counter_config](#)

Structure for configuring a TCC as a counter.

Table 26-13. Members

Type	Name	Description
enum tcc_clock_prescaler	clock_prescaler	Specifies the prescaler value for GCLK_TCC
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
uint32_t	count	Value to initialize the count register
enum tcc_count_direction	direction	Specifies the direction for the TCC to count
enum tcc_count_overflow_dma_trigger_mode	dma_trigger_mode	Counter overflow trigger a DMA request mode
bool	oneshot	When <code>true</code> , the counter will be stopped on the next hardware or software re-trigger event or overflow/underflow
uint32_t	period	Period/top and period/top buffer values for counter
enum tcc_reload_action	reload_action	Specifies the reload or reset time of the counter and prescaler resynchronization on a re-trigger event for the TCC

26.6.2.5. Struct [tcc_events](#)

Event flags for the [tcc_enable_events\(\)](#) and [tcc_disable_events\(\)](#).

Table 26-14. Members

Type	Name	Description
bool	generate_event_on_channel[]	Generate an output event on a channel capture/match. Specify which channels will generate events
bool	generate_event_on_counter_event	Generate an output event on counter boundary. See <code>tcc_event_output_action</code> .
bool	generate_event_on_counter_overflow	Generate an output event on counter overflow/underflow
bool	generate_event_on_counter_retrigger	Generate an output event on counter retrigger
struct tcc_input_event_config	input_config[]	Input events configuration
bool	on_event_perform_channel_action[]	Perform the configured event action when an incoming channel event is signalled
bool	on_input_event_perform_action[]	Perform the configured event action when an incoming event is signalled
struct tcc_output_event_config	output_config	Output event configuration

26.6.2.6. Struct `tcc_input_event_config`

For configuring an input event.

Table 26-15. Members

Type	Name	Description
enum tcc_event_action	action	Event action on incoming event
bool	invert	Invert incoming event input line
bool	modify_action	Modify event action

26.6.2.7. Struct `tcc_match_wave_config`

The structure, which helps to configure a TCC channel for compare operation and wave generation.

Table 26-16. Members

Type	Name	Description
enum tcc_channel_function	channel_function[]	Channel functions selection (capture/match)
uint32_t	match[]	Value to be used for compare match on each channel
enum tcc_wave_generation	wave_generation	Specifies which waveform generation mode to use

Type	Name	Description
enum tcc_wave_polarity	wave_polarity[]	Specifies polarity for match output waveform generation
enum tcc_ramp	wave_ramp	Specifies Ramp mode for waveform generation

26.6.2.8. Struct tcc_module

TCC software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved only for module-internal use.

Table 26-17. Members

Type	Name	Description
tcc_callback_t	callback[]	Array of callbacks
bool	double_buffering_enabled	Set to <code>true</code> to write to buffered registers
uint32_t	enable_callback_mask	Bit mask for callbacks enabled
Tcc *	hw	Hardware module pointer of the associated Timer/Counter peripheral.
uint32_t	register_callback_mask	Bit mask for callbacks registered

26.6.2.9. Struct tcc_non_recoverable_fault_config

Table 26-18. Members

Type	Name	Description
uint8_t	filter_value	Fault filter value applied on TCEx event input line (0x0 ~ 0xF). Must be 0 when TCEx event is used as synchronous event.
enum tcc_fault_state_output	output	Output

26.6.2.10. Struct tcc_output_event_config

Structure used for configuring an output event.

Table 26-19. Members

Type	Name	Description
enum tcc_event_generation_selection	generation_selection	It decides which part of the counter cycle the counter event output is generated
bool	modify_generation_selection	A switch to allow enable/disable of events, without modifying the event output configuration

26.6.2.11. Struct `tcc_pins_config`

Structure which is used when taking wave output from TCC.

Table 26-20. Members

Type	Name	Description
bool	<code>enable_wave_out_pin[]</code>	When <code>true</code> , PWM output pin for the given channel is enabled
uint32_t	<code>wave_out_pin[]</code>	Specifies pin output for each channel
uint32_t	<code>wave_out_pin_mux[]</code>	Specifies MUX setting for each output channel pin

26.6.2.12. Struct `tcc_recoverable_fault_config`

Table 26-21. Members

Type	Name	Description
enum <code>tcc_fault_blanking</code>	<code>blanking</code>	Fault Blanking Start Point for recoverable Fault
uint8_t	<code>blanking_cycles</code>	Fault blanking value (0 ~ 255), disable input source for several TCC clocks after the detection of the waveform edge
enum <code>tcc_fault_capture_action</code>	<code>capture_action</code>	Capture action for recoverable Fault
enum <code>tcc_fault_capture_channel</code>	<code>capture_channel</code>	Channel triggered by recoverable Fault
uint8_t	<code>filter_value</code>	Fault filter value applied on MCEx event input line (0x0 ~ 0xF). Must be 0 when MCEx event is used as synchronous event. Apply to both recoverable and non-recoverable fault.
enum <code>tcc_fault_halt_action</code>	<code>halt_action</code>	Halt action for recoverable Fault
bool	<code>keep</code>	Set to <code>true</code> to enable keep action (keep until end of TCC cycle)
bool	<code>qualification</code>	Set to <code>true</code> to enable input qualification (disable input when output is inactive)
bool	<code>restart</code>	Set to <code>true</code> to enable restart action
enum <code>tcc_fault_source</code>	<code>source</code>	Specifies if the event input generates recoverable Fault. The event system channel connected to MCEx event input must be configured as asynchronous.

26.6.2.13. Struct `tcc_wave_extension_config`

This structure is used to specify the waveform extension features for TCC.

Table 26-22. Members

Type	Name	Description
bool	invert[]	Invert waveform final outputs lines
struct tcc_non_recoverable_fault_config	non_recoverable_fault[]	Configuration for non-recoverable faults
struct tcc_recoverable_fault_config	recoverable_fault[]	Configuration for recoverable faults

26.6.3. Macro Definitions

26.6.3.1. Driver Feature Definition

Define port features set according to different device family.

Macro **FEATURE_TCC_GENERATE_DMA_TRIGGER**

```
#define FEATURE_TCC_GENERATE_DMA_TRIGGER
```

Generate DMA triggers

26.6.3.2. Module Status Flags

TCC status flags, returned by [tcc_get_status\(\)](#) and cleared by [tcc_clear_status\(\)](#).

Macro **TCC_STATUS_CHANNEL_MATCH_CAPTURE**

```
#define TCC_STATUS_CHANNEL_MATCH_CAPTURE(ch)
```

Timer channel *ch* (0 ~ 3) has matched against its compare value, or has captured a new value.

Macro **TCC_STATUS_CHANNEL_OUTPUT**

```
#define TCC_STATUS_CHANNEL_OUTPUT(ch)
```

Timer channel *ch* (0 ~ 3) match/compare output state.

Macro **TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR**

```
#define TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(x)
```

A Non-Recoverable Fault *x* (0 ~ 1) has occurred.

Macro **TCC_STATUS_RECOVERABLE_FAULT_OCCUR**

```
#define TCC_STATUS_RECOVERABLE_FAULT_OCCUR(n)
```

A Recoverable Fault *n* (0 ~ 1 representing A ~ B) has occurred.

Macro **TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT**

```
#define TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT(x)
```

The Non-Recoverable Fault *x* (0 ~ 1) input is present.

Macro **TCC_STATUS_RECOVERABLE_FAULT_PRESENT**

```
#define TCC_STATUS_RECOVERABLE_FAULT_PRESENT(n)
```

A Recoverable Fault *n* (0 ~ 1 representing A ~ B) is present.

Macro **TCC_STATUS_SYNC_READY**

```
#define TCC_STATUS_SYNC_READY
```

Timer registers synchronization has completed, and the synchronized count value may be read.

Macro **TCC_STATUS_CAPTURE_OVERFLOW**

```
#define TCC_STATUS_CAPTURE_OVERFLOW
```

A new value was captured before the previous value was read, resulting in lost data.

Macro **TCC_STATUS_COUNTER_EVENT**

```
#define TCC_STATUS_COUNTER_EVENT
```

A counter event occurred.

Macro **TCC_STATUS_COUNTER_RETRIGGERED**

```
#define TCC_STATUS_COUNTER_RETRIGGERED
```

A counter retrigger occurred.

Macro **TCC_STATUS_COUNT_OVERFLOW**

```
#define TCC_STATUS_COUNT_OVERFLOW
```

The timer count value has overflowed from its maximum value to its minimum when counting upward, or from its minimum value to its maximum when counting downward.

Macro **TCC_STATUS_RAMP_CYCLE_INDEX**

```
#define TCC_STATUS_RAMP_CYCLE_INDEX
```

Ramp period cycle index. In ramp operation, each two period cycles are marked as cycle A and B, the index 0 represents cycle A and 1 represents cycle B.

Macro **TCC_STATUS_STOPPED**

```
#define TCC_STATUS_STOPPED
```

The counter has been stopped (due to disable, stop command, or one-shot).

26.6.3.3. Macro **_TCC_CHANNEL_ENUM_LIST**

```
#define _TCC_CHANNEL_ENUM_LIST(type)
```

Generates table enum list entries for all channels of a given type and channel number on TCC module.

26.6.3.4. Macro **_TCC_ENUM**

```
#define _TCC_ENUM(n, type)
```

Generates a table enum list entry for a given type and index (e.g. "TCC_CALLBACK_MC_CHANNEL_0,").

26.6.3.5. Macro **_TCC_WO_ENUM_LIST**

```
#define _TCC_WO_ENUM_LIST(type)
```

Generates table enum list entries for all output of a given type and waveform output number on TCC module.

26.6.3.6. Macro TCC_NUM_CHANNELS

```
#define TCC_NUM_CHANNELS
```

Maximum number of channels supported by the driver (Channel index from 0 to TCC_NUM_CHANNELS - 1).

26.6.3.7. Macro TCC_NUM_FAULTS

```
#define TCC_NUM_FAULTS
```

Maximum number of (recoverable) faults supported by the driver.

26.6.3.8. Macro TCC_NUM_WAVE_OUTPUTS

```
#define TCC_NUM_WAVE_OUTPUTS
```

Maximum number of wave outputs lines supported by the driver (Output line index from 0 to TCC_NUM_WAVE_OUTPUTS - 1).

26.6.4. Function Definitions

26.6.4.1. Driver Initialization and Configuration

Function tcc_is_syncing()

Determines if the hardware module is currently synchronizing to the bus.

```
bool tcc_is_syncing(  
    const struct tcc_module *const module_inst)
```

Checks to see if the underlying hardware peripheral module is currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Table 26-23. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Returns

Synchronization status of the underlying hardware module.

Table 26-24. Return Values

Return value	Description
false	If the module has completed synchronization
true	If the module synchronization is ongoing

Function `tcc_get_config_defaults()`

Initializes config with predefined default values.

```
void tcc_get_config_defaults(  
    struct tcc_config *const config,  
    Tcc *const hw)
```

This function will initialize a given TCC configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- Don't run in standby
- When setting top, compare, or pattern by API, do double buffering write
- The base timer/counter configurations:
 - GCLK generator 0 clock source
 - No prescaler
 - GCLK reload action
 - Count upward
 - Don't perform one-shot operations
 - Counter starts on 0
 - Period/top value set to maximum
- The match/capture configurations:
 - All Capture compare channel value set to 0
 - No capture enabled (all channels use compare function)
 - Normal frequency wave generation
 - Waveform generation polarity set to 0
 - Don't perform ramp on waveform
- The waveform extension configurations:
 - No recoverable fault is enabled, fault actions are disabled, filter is set to 0
 - No non-recoverable fault state output is enabled and filter is 0
 - No inversion of waveform output
- No channel output enabled
- No PWM pin output enabled
- Pin and MUX configuration not set

Table 26-25. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to a TCC module configuration structure to set
[in]	hw	Pointer to the TCC hardware module

Function `tcc_init()`

Initializes a hardware TCC module instance.

```
enum status_code tcc_init(  
    struct tcc_module *const module_inst,  
    Tcc *const hw,  
    const struct tcc_config *const config)
```

Enables the clock and initializes the given TCC module, based on the given configuration values.

Table 26-26. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	hw	Pointer to the TCC hardware module
[in]	config	Pointer to the TCC configuration options struct

Returns

Status of the initialization procedure.

Table 26-27. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_BUSY	The hardware module was busy when the initialization procedure was attempted
STATUS_INVALID_ARG	An invalid configuration option or argument was supplied
STATUS_ERR_DENIED	The hardware module was already enabled

26.6.4.2. Event Management

Function `tcc_enable_events()`

Enables the TCC module event input or output.

```
enum status_code tcc_enable_events(  
    struct tcc_module *const module_inst,  
    struct tcc_events *const events)
```

Enables one or more input or output events to or from the TCC module. See [tcc_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 26-28. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	events	Struct containing flags of events to enable or configure

Returns

Status of the events setup procedure.

Table 26-29. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_INVALID_ARG	An invalid configuration option or argument was supplied

Function tcc_disable_events()

Disables the event input or output of a TCC instance.

```
void tcc_disable_events(
    struct tcc_module *const module_inst,
    struct tcc_events *const events)
```

Disables one or more input or output events for the given TCC module. See [tcc_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 26-30. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	events	Struct containing flags of events to disable

26.6.4.3. Enable/Disable/Reset**Function tcc_enable()**

Enable the TCC module.

```
void tcc_enable(
    const struct tcc_module *const module_inst)
```

Enables a TCC module that has been previously initialized. The counter will start when the counter is enabled.

Note: When the counter is configured to re-trigger on an event, the counter will not start until the next incoming event appears. Then it restarts on any following event.

Table 26-31. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Function tcc_disable()

Disables the TCC module.

```
void tcc_disable(
    const struct tcc_module *const module_inst)
```

Disables a TCC module and stops the counter.

Table 26-32. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Function tcc_reset()

Resets the TCC module.

```
void tcc_reset(
    const struct tcc_module *const module_inst)
```

Resets the TCC module, restoring all hardware module registers to their default values and disabling the module. The TCC module will not be accessible while the reset is being performed.

Note: When resetting a 32-bit counter only the master TCC module's instance structure should be passed to the function.

Table 26-33. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

26.6.4.4. Set/Toggle Count Direction**Function tcc_set_count_direction()**

Sets the TCC module count direction.

```
void tcc_set_count_direction(
    const struct tcc_module *const module_inst,
    enum tcc_count_direction dir)
```

Sets the count direction of an initialized TCC module. The specified TCC module can remain running or stopped.

Table 26-34. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	dir	New timer count direction to set

Function tcc_toggle_count_direction()

Toggles the TCC module count direction.

```
void tcc_toggle_count_direction(
    const struct tcc_module *const module_inst)
```

Toggles the count direction of an initialized TCC module. The specified TCC module can remain running or stopped.

Table 26-35. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

26.6.4.5. Get/Set Count Value

Function `tcc_get_count_value()`

Get count value of the given TCC module.

```
uint32_t tcc_get_count_value(  
    const struct tcc_module *const module_inst)
```

Retrieves the current count value of a TCC module. The specified TCC module can remain running or stopped.

Table 26-36. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Returns

Count value of the specified TCC module.

Function `tcc_set_count_value()`

Sets count value for the given TCC module.

```
enum status_code tcc_set_count_value(  
    const struct tcc_module *const module_inst,  
    const uint32_t count)
```

Sets the timer count value of an initialized TCC module. The specified TCC module can remain running or stopped.

Table 26-37. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	count	New timer count value to set

Returns

Status which indicates whether the new value is set.

Table 26-38. Return Values

Return value	Description
STATUS_OK	The timer count was updated successfully
STATUS_ERR_INVALID_ARG	An invalid timer counter size was specified

26.6.4.6. Stop/Restart Counter

Function `tcc_stop_counter()`

Stops the counter.

```
void tcc_stop_counter(  
    const struct tcc_module *const module_inst)
```

This function will stop the counter. When the counter is stopped the value in the count register is set to 0 if the counter was counting up, or maximum or the top value if the counter was counting down.

Table 26-39. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

Function `tcc_restart_counter()`

Starts the counter from beginning.

```
void tcc_restart_counter(
    const struct tcc_module *const module_inst)
```

Restarts an initialized TCC module's counter.

Table 26-40. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

26.6.4.7. Generate TCC DMA Triggers Command

Function `tcc_dma_trigger_command()`

TCC DMA Trigger.

```
void tcc_dma_trigger_command(
    const struct tcc_module *const module_inst)
```

TCC DMA trigger command.

Table 26-41. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct

26.6.4.8. Get/Set Compare/Capture Register

Function `tcc_get_capture_value()`

Gets the TCC module capture value.

```
uint32_t tcc_get_capture_value(
    const struct tcc_module *const module_inst,
    const enum tcc_match_capture_channel channel_index)
```

Retrieves the capture value in the indicated TCC module capture channel.

Table 26-42. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	channel_index	Index of the Compare Capture channel to read

Returns

Capture value stored in the specified timer channel.

Function `tcc_set_compare_value()`

Sets a TCC module compare value.

```
enum status_code tcc_set_compare_value(  
    const struct tcc_module *const module_inst,  
    const enum tcc_match_capture_channel channel_index,  
    const uint32_t compare)
```

Writes a compare value to the given TCC module compare/capture channel.

If double buffering is enabled it always write to the buffer register. The value will then be updated immediately by calling `tcc_force_double_buffer_update()`, or be updated when the lock update bit is cleared and the UPDATE condition happen.

Table 26-43. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	channel_index	Index of the compare channel to write to
[in]	compare	New compare value to set

Returns

Status of the compare update procedure.

Table 26-44. Return Values

Return value	Description
STATUS_OK	The compare value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied or compare value exceed resolution

26.6.4.9. Set Top Value

Function `tcc_set_top_value()`

Set the timer TOP/PERIOD value.

```
enum status_code tcc_set_top_value(  
    const struct tcc_module *const module_inst,  
    const uint32_t top_value)
```

This function writes the given value to the PER/PERB register.

If double buffering is enabled it always write to the buffer register (PERB). The value will then be updated immediately by calling `tcc_force_double_buffer_update()`, or be updated when the lock update bit is cleared and the UPDATE condition happen.

When using MFRQ, the top value is defined by the CC0 register value and the PER value is ignored, so `tcc_set_compare_value` (module,channel_0,value) must be used instead of this function to change the actual top value in that case. For all other waveforms operation the top value is defined by PER register value.

Table 26-45. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	top_value	New value to be loaded into the PER/PERB register

Returns

Status of the TOP set procedure.

Table 26-46. Return Values

Return value	Description
STATUS_OK	The timer TOP value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied or top/period value exceed resolution

26.6.4.10. Set Output Pattern**Function `tcc_set_pattern()`**

Sets the TCC module waveform output pattern.

```
enum status_code tcc_set_pattern(
    const struct tcc_module *const module_inst,
    const uint32_t line_index,
    const enum tcc_output_pattern pattern)
```

Force waveform output line to generate specific pattern (0, 1, or as is).

If double buffering is enabled it always write to the buffer register. The value will then be updated immediately by calling `tcc_force_double_buffer_update()`, or be updated when the lock update bit is cleared and the UPDATE condition happen.

Table 26-47. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	line_index	Output line index
[in]	pattern	Output pattern to use (tcc_output_pattern)

Returns

Status of the pattern set procedure.

Table 26-48. Return Values

Return value	Description
STATUS_OK	The PATT register is updated successfully
STATUS_ERR_INVALID_ARG	An invalid line index was supplied

26.6.4.11. Set Ramp Index

Function `tcc_set_ramp_index()`

Sets the TCC module ramp index on next cycle.

```
void tcc_set_ramp_index(  
    const struct tcc_module *const module_inst,  
    const enum tcc_ramp_index ramp_index)
```

In RAMP2 and RAMP2A operation, we can force either cycle A or cycle B at the output, on the next clock cycle. When ramp index command is disabled, cycle A and cycle B will appear at the output, on alternate clock cycles. See [tcc_ramp](#).

Table 26-49. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	ramp_index	Ramp index (tcc_ramp_index) of the next cycle

26.6.4.12. Status Management

Function `tcc_is_running()`

Checks if the timer/counter is running.

```
bool tcc_is_running(  
    struct tcc_module *const module_inst)
```

Table 26-50. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Returns

Status which indicates whether the module is running.

Table 26-51. Return Values

Return value	Description
true	The timer/counter is running
false	The timer/counter is stopped

Function `tcc_get_status()`

Retrieves the current module status.

```
uint32_t tcc_get_status(  
    struct tcc_module *const module_inst)
```

Retrieves the status of the module, giving overall state information.

Table 26-52. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Returns

Bitmask of TCC_STATUS_* flags.

Table 26-53. Return Values

Return value	Description
TCC_STATUS_CHANNEL_MATCH_CAPTURE(n)	Channel n match/capture has occurred
TCC_STATUS_CHANNEL_OUTPUT(n)	Channel n match/capture output state
TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(x)	Non-recoverable fault x has occurred
TCC_STATUS_RECOVERABLE_FAULT_OCCUR(n)	Recoverable fault n has occurred
TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT(x)	Non-recoverable fault x input present
TCC_STATUS_RECOVERABLE_FAULT_PRESENT(n)	Recoverable fault n input present
TCC_STATUS_SYNC_READY	None of register is syncing
TCC_STATUS_CAPTURE_OVERFLOW	Timer capture data has overflowed
TCC_STATUS_COUNTER_EVENT	Timer counter event has occurred
TCC_STATUS_COUNT_OVERFLOW	Timer count value has overflowed
TCC_STATUS_COUNTER_RETRIGGERED	Timer counter has been retriggered
TCC_STATUS_STOP	Timer counter has been stopped
TCC_STATUS_RAMP_CYCLE_INDEX	Wave ramp index for cycle

Function tcc_clear_status()

Clears a module status flag.

```
void tcc_clear_status(
    struct tcc_module *const module_inst,
    const uint32_t status_flags)
```

Clears the given status flag of the module.

Table 26-54. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct
[in]	status_flags	Bitmask of TCC_STATUS_* flags to clear

26.6.4.13. Double Buffering Management

Function `tcc_enable_double_buffering()`

Enable TCC double buffering write.

```
void tcc_enable_double_buffering(  
    struct tcc_module *const module_inst)
```

When double buffering write is enabled, the following function will write values to buffered registers instead of effective ones (buffered):

- PERB: through `tcc_set_top_value()`
- CCBx(x is 0~3): through `tcc_set_compare_value()`
- PATTB: through `tcc_set_pattern()`

Then, on UPDATE condition the buffered registers are committed to regular ones to take effect.

Table 26-55. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function `tcc_disable_double_buffering()`

Disable TCC double buffering Write.

```
void tcc_disable_double_buffering(  
    struct tcc_module *const module_inst)
```

When double buffering write is disabled, following function will write values to effective registers (not buffered):

- PER: through `tcc_set_top_value()`
- CCx(x is 0~3): through `tcc_set_compare_value()`
- PATT: through `tcc_set_pattern()`

Note: This function does not lock double buffer update, which means on next UPDATE condition the last written buffered values will be committed to take effect. Invoke `tcc_lock_double_buffer_update()` before this function to disable double buffering update, if this change is not expected.

Table 26-56. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function `tcc_lock_double_buffer_update()`

Lock the TCC double buffered registers updates.

```
void tcc_lock_double_buffer_update(  
    struct tcc_module *const module_inst)
```

Locks the double buffered registers so they will not be updated through their buffered values on UPDATE conditions.

Table 26-57. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function tcc_unlock_double_buffer_update()

Unlock the TCC double buffered registers updates.

```
void tcc_unlock_double_buffer_update(
    struct tcc_module *const module_inst)
```

Unlock the double buffered registers so they will be updated through their buffered values on UPDATE conditions.

Table 26-58. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function tcc_force_double_buffer_update()

Force the TCC double buffered registers to update once.

```
void tcc_force_double_buffer_update(
    struct tcc_module *const module_inst)
```

Table 26-59. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function tcc_enable_circular_buffer_top()

Enable Circular option for double buffered Top/Period Values.

```
void tcc_enable_circular_buffer_top(
    struct tcc_module *const module_inst)
```

Enable circular option for the double buffered top/period values. On each UPDATE condition, the contents of PERB and PER are switched, meaning that the contents of PERB are transferred to PER and the contents of PER are transferred to PERB.

Table 26-60. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function tcc_disable_circular_buffer_top()

Disable Circular option for double buffered Top/Period Values.

```
void tcc_disable_circular_buffer_top(
    struct tcc_module *const module_inst)
```

Stop circularing the double buffered top/period values.

Table 26-61. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct

Function `tcc_set_double_buffer_top_values()`

Set the timer TOP/PERIOD value and buffer value.

```
enum status_code tcc_set_double_buffer_top_values(
    const struct tcc_module *const module_inst,
    const uint32_t top_value,
    const uint32_t top_buffer_value)
```

This function writes the given value to the PER and PERB register. Usually as preparation for double buffer or circuled double buffer (circular buffer).

When using MFRQ, the top values are defined by the CC0 and CCB0, the PER and PERB values are ignored, so [tcc_set_double_buffer_compare_values](#) (module,channel_0,value,buffer) must be used instead of this function to change the actual top values in that case. For all other waveforms operation the top values are defined by PER and PERB registers values.

Table 26-62. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	top_value	New value to be loaded into the PER register
[in]	top_buffer_value	New value to be loaded into the PERB register

Returns

Status of the TOP set procedure.

Table 26-63. Return Values

Return value	Description
STATUS_OK	The timer TOP value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied or top/period value exceed resolution

Function `tcc_enable_circular_buffer_compare()`

Enable circular option for double buffered compare values.

```
enum status_code tcc_enable_circular_buffer_compare(
    struct tcc_module *const module_inst,
    enum tcc_match_capture_channel channel_index)
```

Enable circular option for the double buffered channel compare values. On each UPDATE condition, the contents of CCBx and CCx are switched, meaning that the contents of CCBx are transferred to CCx and the contents of CCx are transferred to CCBx.

Table 26-64. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct
[in]	channel_index	Index of the compare channel to set up to

Table 26-65. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_INVALID_ARG	An invalid channel index is supplied

Function tcc_disable_circular_buffer_compare()

Disable circular option for double buffered compare values.

```
enum status_code tcc_disable_circular_buffer_compare(
    struct tcc_module *const module_inst,
    enum tcc_match_capture_channel channel_index)
```

Stop circularing the double buffered compare values.

Table 26-66. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct
[in]	channel_index	Index of the compare channel to set up to

Table 26-67. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_INVALID_ARG	An invalid channel index is supplied

Function tcc_set_double_buffer_compare_values()

Sets a TCC module compare value and buffer value.

```
enum status_code tcc_set_double_buffer_compare_values(
    struct tcc_module *const module_inst,
    enum tcc_match_capture_channel channel_index,
    const uint32_t compare,
    const uint32_t compare_buffer)
```

Writes compare value and buffer to the given TCC module compare/capture channel. Usually as preparation for double buffer or circularized double buffer (circular buffer).

Table 26-68. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	channel_index	Index of the compare channel to write to
[in]	compare	New compare value to set
[in]	compare_buffer	New compare buffer value to set

Returns

Status of the compare update procedure.

Table 26-69. Return Values

Return value	Description
STATUS_OK	The compare value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied or compare value exceed resolution

26.6.5. Enumeration Definitions**26.6.5.1. Enum tcc_callback**

Enum for the possible callback types for the TCC module.

Table 26-70. Members

Enum value	Description
TCC_CALLBACK_OVERFLOW	Callback for TCC overflow
TCC_CALLBACK_RETRIGGER	Callback for TCC Retrigger
TCC_CALLBACK_COUNTER_EVENT	Callback for TCC counter event
TCC_CALLBACK_ERROR	Callback for capture overflow error
TCC_CALLBACK_FAULTA	Callback for Recoverable Fault A
TCC_CALLBACK_FAULTB	Callback for Recoverable Fault B
TCC_CALLBACK_FAULT0	Callback for Non-Recoverable Fault 0
TCC_CALLBACK_FAULT1	Callback for Non-Recoverable Fault 1
TCC_CALLBACK_CHANNEL_n	Channel callback type table for TCC Each TCC module may contain several callback types for channels; each channel will have its own callback type in the table, with the channel index number substituted for "n" in the channel callback type (e.g. TCC_MATCH_CAPTURE_CHANNEL_0).

26.6.5.2. Enum `tcc_channel_function`

To set a timer channel either in compare or in capture mode.

Table 26-71. Members

Enum value	Description
<code>TCC_CHANNEL_FUNCTION_COMPARE</code>	TCC channel performs compare operation
<code>TCC_CHANNEL_FUNCTION_CAPTURE</code>	TCC channel performs capture operation

26.6.5.3. Enum `tcc_clock_prescaler`

This enum is used to choose the clock prescaler configuration. The prescaler divides the clock frequency of the TCC module to operate TCC at a slower clock rate.

Table 26-72. Members

Enum value	Description
<code>TCC_CLOCK_PRESCALER_DIV1</code>	Divide clock by 1
<code>TCC_CLOCK_PRESCALER_DIV2</code>	Divide clock by 2
<code>TCC_CLOCK_PRESCALER_DIV4</code>	Divide clock by 4
<code>TCC_CLOCK_PRESCALER_DIV8</code>	Divide clock by 8
<code>TCC_CLOCK_PRESCALER_DIV16</code>	Divide clock by 16
<code>TCC_CLOCK_PRESCALER_DIV64</code>	Divide clock by 64
<code>TCC_CLOCK_PRESCALER_DIV256</code>	Divide clock by 256
<code>TCC_CLOCK_PRESCALER_DIV1024</code>	Divide clock by 1024

26.6.5.4. Enum `tcc_count_direction`

Used when selecting the Timer/Counter count direction.

Table 26-73. Members

Enum value	Description
<code>TCC_COUNT_DIRECTION_UP</code>	Timer should count upward
<code>TCC_COUNT_DIRECTION_DOWN</code>	Timer should count downward

26.6.5.5. Enum `tcc_count_overflow_dma_trigger_mode`

Used when selecting the Timer/Counter overflow DMA request mode.

Table 26-74. Members

Enum value	Description
TCC_COUNT_OVERFLOW_DMA_TRIGGER_MODE_CONTINUE	TCC generates a DMA request on each cycle when an update condition is detected
TCC_COUNT_OVERFLOW_DMA_TRIGGER_MODE_ONE_SHOT	When an update condition is detected, the TCC generates a DMA trigger on the cycle following the DMA One-Shot Command written to the Control B register

26.6.5.6. Enum tcc_event0_action

Event action to perform when the module is triggered by event0.

Table 26-75. Members

Enum value	Description
TCC_EVENT0_ACTION_OFF	No event action
TCC_EVENT0_ACTION_RETRIGGER	Re-trigger Counter on event
TCC_EVENT0_ACTION_COUNT_EVENT	Count events (increment or decrement, depending on count direction)
TCC_EVENT0_ACTION_START	Start counter on event
TCC_EVENT0_ACTION_INCREMENT	Increment counter on event
TCC_EVENT0_ACTION_COUNT_DURING_ACTIVE	Count during active state of asynchronous event
TCC_EVENT0_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event

26.6.5.7. Enum tcc_event1_action

Event action to perform when the module is triggered by event1.

Table 26-76. Members

Enum value	Description
TCC_EVENT1_ACTION_OFF	No event action
TCC_EVENT1_ACTION_RETRIGGER	Re-trigger Counter on event
TCC_EVENT1_ACTION_DIR_CONTROL	The event source must be an asynchronous event, and the input value will override the direction settings. If TCEINVx is 0 and input event is LOW: counter will count up. If TCEINVx is 0 and input event is HIGH: counter will count down.

Enum value	Description
TCC_EVENT1_ACTION_STOP	Stop counter on event
TCC_EVENT1_ACTION_DECREMENT	Decrement on event
TCC_EVENT1_ACTION_PERIOD_PULSE_WIDTH_CAPTURE	Store period in capture register 0, pulse width in capture register 1
TCC_EVENT1_ACTION_PULSE_WIDTH_PERIOD_CAPTURE	Store pulse width in capture register 0, period in capture register 1
TCC_EVENT1_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event

26.6.5.8. Enum `tcc_event_action`

Event action to perform when the module is triggered by events.

Table 26-77. Members

Enum value	Description
TCC_EVENT_ACTION_OFF	No event action
TCC_EVENT_ACTION_STOP	Stop counting, the counter will maintain its current value, waveforms are set to a defined Non-Recoverable State output (<code>tcc_non_recoverable_state_output</code>).
TCC_EVENT_ACTION_RETRIGGER	Re-trigger counter on event, may generate an event if the re-trigger event output is enabled. Note: When re-trigger event action is enabled, enabling the counter will not start until the next incoming event appears.
TCC_EVENT_ACTION_START	Start counter when previously stopped. Start counting on the event rising edge. Further events will not restart the counter; the counter keeps on counting using prescaled <code>GCLK_TCCx</code> , until it reaches TOP or Zero depending on the direction.
TCC_EVENT_ACTION_COUNT_EVENT	Count events; i.e. Increment or decrement depending on count direction.

Enum value	Description
TCC_EVENT_ACTION_DIR_CONTROL	The event source must be an asynchronous event, input value will overrides the direction settings (input low: counting up, input high: counting down).
TCC_EVENT_ACTION_INCREMENT	Increment the counter on event, irrespective of count direction
TCC_EVENT_ACTION_DECREMENT	Decrement the counter on event, irrespective of count direction
TCC_EVENT_ACTION_COUNT_DURING_ACTIVE	Count during active state of asynchronous event. In this case, depending on the count direction, the count will be incremented or decremented on each prescaled GCLK_TCCx, as long as the input event remains active.
TCC_EVENT_ACTION_PERIOD_PULSE_WIDTH_CAPTURE	Store period in capture register 0, pulse width in capture register 1
TCC_EVENT_ACTION_PULSE_WIDTH_PERIOD_CAPTURE	Store pulse width in capture register 0, period in capture register 1
TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event

26.6.5.9. Enum tcc_event_generation_selection

This enum is used to define the point at which the counter event is generated.

Table 26-78. Members

Enum value	Description
TCC_EVENT_GENERATION_SELECTION_START	Counter Event is generated when a new counter cycle starts
TCC_EVENT_GENERATION_SELECTION_END	Counter Event is generated when a counter cycle ends
TCC_EVENT_GENERATION_SELECTION_BETWEEN	Counter Event is generated when a counter cycle ends, except for the first and last cycles
TCC_EVENT_GENERATION_SELECTION_BOUNDARY	Counter Event is generated when a new counter cycle starts or ends

26.6.5.10. Enum tcc_fault_blanking

Table 26-79. Members

Enum value	Description
TCC_FAULT_BLANKING_DISABLE	No blanking
TCC_FAULT_BLANKING_RISING_EDGE	Blanking applied from rising edge of the output waveform
TCC_FAULT_BLANKING_FALLING_EDGE	Blanking applied from falling edge of the output waveform
TCC_FAULT_BLANKING_BOTH_EDGE	Blanking applied from each toggle of the output waveform

26.6.5.11. Enum tcc_fault_capture_action

Table 26-80. Members

Enum value	Description
TCC_FAULT_CAPTURE_DISABLE	Capture disabled
TCC_FAULT_CAPTURE_EACH	Capture on Fault, each value is captured
TCC_FAULT_CAPTURE_MINIMUM	Capture the minimum detection, but notify on smaller ones
TCC_FAULT_CAPTURE_MAXIMUM	Capture the maximum detection, but notify on bigger ones
TCC_FAULT_CAPTURE_SMALLER	Capture if the value is smaller than last, notify event or interrupt if previous stamp is confirmed to be "local minimum" (not bigger than current stamp).
TCC_FAULT_CAPTURE_BIGGER	Capture if the value is bigger than last, notify event or interrupt if previous stamp is confirmed to be "local maximum" (not smaller than current stamp).
TCC_FAULT_CAPTURE_CHANGE	Capture if the time stamps changes its increment direction

26.6.5.12. Enum tcc_fault_capture_channel

Table 26-81. Members

Enum value	Description
TCC_FAULT_CAPTURE_CHANNEL_0	Recoverable fault triggers channel 0 capture operation
TCC_FAULT_CAPTURE_CHANNEL_1	Recoverable fault triggers channel 1 capture operation
TCC_FAULT_CAPTURE_CHANNEL_2	Recoverable fault triggers channel 2 capture operation
TCC_FAULT_CAPTURE_CHANNEL_3	Recoverable fault triggers channel 3 capture operation

26.6.5.13. Enum tcc_fault_halt_action

Table 26-82. Members

Enum value	Description
TCC_FAULT_HALT_ACTION_DISABLE	Halt action disabled.
TCC_FAULT_HALT_ACTION_HW_HALT	Hardware halt action, counter is halted until restart
TCC_FAULT_HALT_ACTION_SW_HALT	Software halt action, counter is halted until fault bit cleared
TCC_FAULT_HALT_ACTION_NON_RECOVERABLE	Non-Recoverable fault, force output to pre-defined level

26.6.5.14. Enum tcc_fault_keep

Table 26-83. Members

Enum value	Description
TCC_FAULT_KEEP_DISABLE	Disable keeping, wave output released as soon as fault is released
TCC_FAULT_KEEP_TILL_END	Keep wave output until end of TCC cycle

26.6.5.15. Enum tcc_fault_qualification

Table 26-84. Members

Enum value	Description
TCC_FAULT_QUALIFICATION_DISABLE	The input is not disabled on compare condition
TCC_FAULT_QUALIFICATION_BY_OUTPUT	The input is disabled when match output signal is at inactive level

26.6.5.16. Enum tcc_fault_restart

Table 26-85. Members

Enum value	Description
TCC_FAULT_RESTART_DISABLE	Restart Action disabled
TCC_FAULT_RESTART_ENABLE	Restart Action enabled

26.6.5.17. Enum tcc_fault_source

Table 26-86. Members

Enum value	Description
TCC_FAULT_SOURCE_DISABLE	Fault input is disabled
TCC_FAULT_SOURCE_ENABLE	Match Capture Event x (x=0,1) input

Enum value	Description
TCC_FAULT_SOURCE_INVERT	Inverted MCEx (x=0,1) event input
TCC_FAULT_SOURCE_ALTFAULT	Alternate fault (A or B) state at the end of the previous period

26.6.5.18. Enum tcc_fault_state_output

Table 26-87. Members

Enum value	Description
TCC_FAULT_STATE_OUTPUT_OFF	Non-recoverable fault output is tri-stated
TCC_FAULT_STATE_OUTPUT_0	Non-recoverable fault force output 0
TCC_FAULT_STATE_OUTPUT_1	Non-recoverable fault force output 1

26.6.5.19. Enum tcc_match_capture_channel

This enum is used to specify which capture/match channel to do operations on.

Table 26-88. Members

Enum value	Description
TCC_MATCH_CAPTURE_CHANNEL_n	Match capture channel index table for TCC Each TCC module may contain several match capture channels; each channel will have its own index in the table, with the index number substituted for "n" in the index name (e.g. TCC_MATCH_CAPTURE_CHANNEL_0).

26.6.5.20. Enum tcc_output_inversion

Used when enabling or disabling output inversion.

Table 26-89. Members

Enum value	Description
TCC_OUTPUT_INVERSION_DISABLE	Output inversion not to be enabled
TCC_OUTPUT_INVERSION_ENABLE	Invert the output from WO[x]

26.6.5.21. Enum tcc_output_pattern

Used when disabling output pattern or when selecting a specific pattern.

Table 26-90. Members

Enum value	Description
TCC_OUTPUT_PATTERN_DISABLE	SWAP output pattern is not used
TCC_OUTPUT_PATTERN_0	Pattern 0 is applied to SWAP output
TCC_OUTPUT_PATTERN_1	Pattern 1 is applied to SWAP output

26.6.5.22. Enum `tcc_ramp`

Ramp operations which are supported in single-slope PWM generation.

Table 26-91. Members

Enum value	Description
TCC_RAMP_RAMP1	Default timer/counter PWM operation
TCC_RAMP_RAMP2A	Uses a single channel (CC0) to control both CC0/CC1 compare outputs. In cycle A, the channel 0 output is disabled, and in cycle B, the channel 1 output is disabled.
TCC_RAMP_RAMP2	Uses channels CC0 and CC1 to control compare outputs. In cycle A, the channel 0 output is disabled, and in cycle B, the channel 1 output is disabled.

26.6.5.23. Enum `tcc_ramp_index`

In ramp operation, each two period cycles are marked as cycle A and B, the index 0 represents cycle A and 1 represents cycle B.

Table 26-92. Members

Enum value	Description
TCC_RAMP_INDEX_DEFAULT	Default, cycle index toggles.
TCC_RAMP_INDEX_FORCE_B	Force next cycle to be cycle B (set to 1)
TCC_RAMP_INDEX_FORCE_A	Force next cycle to be cycle A (clear to 0)
TCC_RAMP_INDEX_FORCE_KEEP	Force next cycle keeping the same as current

26.6.5.24. Enum `tcc_reload_action`

This enum specifies how the counter is reloaded and whether the prescaler should be restarted.

Table 26-93. Members

Enum value	Description
TCC_RELOAD_ACTION_GCLK	The counter is reloaded/reset on the next GCLK and starts counting on the prescaler clock
TCC_RELOAD_ACTION_PRESC	The counter is reloaded/reset on the next prescaler clock
TCC_RELOAD_ACTION_RESYNC	The counter is reloaded/reset on the next GCLK, and the prescaler is restarted as well

26.6.5.25. Enum `tcc_wave_generation`

This enum is used to specify the waveform generation mode.

Table 26-94. Members

Enum value	Description
TCC_WAVE_GENERATION_NORMAL_FREQ	Normal Frequency: Top is the PER register, output toggled on each compare match
TCC_WAVE_GENERATION_MATCH_FREQ	Match Frequency: Top is CC0 register, output toggles on each update condition
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM	Single-Slope PWM: Top is the PER register, CCx controls duty cycle (output active when count is greater than CCx)
TCC_WAVE_GENERATION_DOUBLE_SLOPE_CRITICAL	Double-slope (count up and down), non centre-aligned: Top is the PER register, CC[x] controls duty cycle while counting up and CC[x+N/2] controls it while counting down
TCC_WAVE_GENERATION_DOUBLE_SLOPE_BOTTOM	Double-slope (count up and down), interrupt/event at Bottom (Top is the PER register, output active when count is greater than CCx)
TCC_WAVE_GENERATION_DOUBLE_SLOPE_BOTH	Double-slope (count up and down), interrupt/event at Bottom and Top: (Top is the PER register, output active when count is lower than CCx)
TCC_WAVE_GENERATION_DOUBLE_SLOPE_TOP	Double-slope (count up and down), interrupt/event at Top (Top is the PER register, output active when count is greater than CCx)

26.6.5.26. Enum tcc_wave_output

This enum is used to specify which wave output to do operations on.

Table 26-95. Members

Enum value	Description
TCC_WAVE_OUTPUT_n	Waveform output index table for TCC Each TCC module may contain several wave outputs; each output will have its own index in the table, with the index number substituted for "n" in the index name (e.g. TCC_WAVE_OUTPUT_0).

26.6.5.27. Enum tcc_wave_polarity

Specifies whether the wave output needs to be inverted or not.

Table 26-96. Members

Enum value	Description
TCC_WAVE_POLARITY_0	Wave output is not inverted
TCC_WAVE_POLARITY_1	Wave output is inverted

26.7. Extra Information for TCC Driver

26.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
DMA	Direct Memory Access
TCC	Timer Counter for Control Applications
PWM	Pulse Width Modulation
PWP	Pulse Width Period
PPW	Period Pulse Width

26.7.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

26.7.3. Errata

There are no errata related to this driver.

26.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Add double buffering functionality
Add fault handling functionality
Initial Release

26.8. Examples for TCC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Timer Counter for Control Applications \(TCC\) Driver](#). QSGs are simple examples with step-by-step instructions to

configure and use this driver in a selection of use cases. Note that QSGs can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for TCC - Basic](#)
- [Quick Start Guide for TCC - Double Buffering and Circular](#)
- [Quick Start Guide for TCC - Timer](#)
- [Quick Start Guide for TCC - Callback](#)
- [Quick Start Guide for TCC - Non-Recoverable Fault](#)
- [Quick Start Guide for TCC - Recoverable Fault](#)
- [Quick Start Guide for Using DMA with TCC](#)

26.8.1. Quick Start Guide for TCC - Basic

The supported board list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal. Here the pulse width is set to one quarter of the period. When the PWM signal connects to LED, LED will light. To see the waveform, you may need an oscilloscope.

The PWM output is set up as follows:

Board	Pin	Connect to
SAM D21 Xpro	PB30	LED0
SAM R21 Xpro	PA19	LED0
SAM L21 Xpro	PB10	LED0
SAM L22 Xpro	PC27	LED0
SAM DA1 Xpro	PB30	LED0
SAM C21 Xpro	PA15	LED0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- Prescaler is set to 256
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No fault or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action

- No event generation enabled
- Counter starts on 0
- Counter top set to 0xFFFF
- Capture compare channel 0 set to 0xFFFF/4

26.8.1.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

```
#if SAMR30

#define CONF_PWM_MODULE      LED_0_PWM3CTRL_MODULE
#define CONF_PWM_CHANNEL    LED_0_PWM3CTRL_CHANNEL
#define CONF_PWM_OUTPUT     LED_0_PWM3CTRL_OUTPUT
#define CONF_PWM_OUT_PIN    LED_0_PWM3CTRL_PIN
#define CONF_PWM_OUT_MUX    LED_0_PWM3CTRL_MUX
#else

#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL    LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT     LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN    LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX    LED_0_PWM4CTRL_MUX
#endif
```

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

Copy-paste the following setup code to your user application:

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV256;
    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = (0xFFFF / 4);

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
}
```

```
tcc_enable(&tcc_instance);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_tcc();
```

Workflow

1. Create a module software instance structure for the TCC module to store the TCC driver state while it is in use.

```
struct tcc_module tcc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TCC module.

1. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```
struct tcc_config config_tcc;
```

2. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TCC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```
config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV256;
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = (0xFFFF / 4);
```

4. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;
```

5. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

6. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

26.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {
    /* Infinite loop */
}
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TCC module.

```
while (true) {  
    /* Infinite loop */  
}
```

26.8.2. Quick Start Guide for TCC - Double Buffering and Circular

The supported board list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal. Here the pulse width alters in one quarter and three quarter of the period. When the PWM signal connects to LED, LED will light. To see the waveform, you may need an oscilloscope.

The PWM output is set up as follows:

Board	Pin	Connect to
SAM D21 Xpro	PB30	LED0
SAM R21 Xpro	PA19	LED0
SAM L21 Xpro	PB10	LED0
SAM L22 Xpro	PC27	LED0
SAM DA1 Xpro	PB30	LED0
SAM C21 Xpro	PA15	LED0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- Prescaler is set to 1024
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No fault or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Counter top set to 8000
- Capture compare channel set to 8000/4

- Capture compare channel buffer set to $8000 \times 3/4$
- Circular option for compare channel is enabled so that the compare values keep switching on update condition

26.8.2.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

```
#if SAMR30

#define CONF_PWM_MODULE      LED_0_PWM3CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM3CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM3CTRL_OUTPUT

#define CONF_PWM_OUT_PIN     LED_0_PWM3CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM3CTRL_MUX
#else

#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
#endif
```

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

Copy-paste the following setup code to your user application:

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV1024;
    config_tcc.counter.period = 8000;
    config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = (8000 / 4);

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    tcc_set_compare_value(&tcc_instance,
        (enum tcc_match_capture_channel)CONF_PWM_CHANNEL, 8000*3/4);
}
```



```
tcc_enable_circular_buffer_compare(&tcc_instance,
    (enum tcc_match_capture_channel)CONF_PWM_CHANNEL);

tcc_enable(&tcc_instance);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_tcc();
```

Workflow

1. Create a module software instance structure for the TCC module to store the TCC driver state while it is in use.

```
struct tcc_module tcc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TCC module.

1. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```
struct tcc_config config_tcc;
```

2. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TCC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```
config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV1024;
config_tcc.counter.period = 8000;
config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = (8000 / 4);
```

4. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;
```

5. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

6. Set to compare buffer value and enable circular of double buffered compare values.

```
tcc_set_compare_value(&tcc_instance,
    (enum tcc_match_capture_channel)CONF_PWM_CHANNEL,
    8000*3/4);
tcc_enable_circular_buffer_compare(&tcc_instance,
    (enum tcc_match_capture_channel)CONF_PWM_CHANNEL);
```

7. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

26.8.2.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {  
    /* Infinite loop */  
}
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TCC module.

```
while (true) {  
    /* Infinite loop */  
}
```

26.8.3. Quick Start Guide for TCC - Timer

The supported board list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro
- SAM D11 Xplained Pro

In this use case, the TCC will be used as a timer, to generate overflow and compare match callbacks. In the callbacks the on-board LED is toggled.

The TCC module will be set up as follows:

- GCLK generator 1 (GCLK 32K) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- Prescaler is divided by 64
- GCLK reload action
- Count upward
- Don't run in standby
- No waveform outputs
- No capture enabled
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Counter top set to 2000 (about 4s) and generate overflow callback
- Channel 0 is set to compare and match value 900 and generate callback
- Channel 1 is set to compare and match value 930 and generate callback
- Channel 2 is set to compare and match value 1100 and generate callback
- Channel 3 is set to compare and match value 1250 and generate callback

26.8.3.1. Quick Start

Prerequisites

For this use case, XOSC32K should be enabled and available through GCLK generator 1 clock source selection. Within Atmel Software Framework (ASF) it can be done through modifying *conf_clocks.h*. See System Clock Management Driver for more details about clock configuration.

Code

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

Copy-paste the following callback function code to your user application:

```
static void tcc_callback_to_toggle_led(  
    struct tcc_module *const module_inst)  
{  
    port_pin_toggle_output_level(LED0_PIN);  
}
```

Copy-paste the following setup code to your user application:

```
static void configure_tcc(void)  
{  
    struct tcc_config config_tcc;  
    tcc_get_config_defaults(&config_tcc, TCC0);  
  
    config_tcc.counter.clock_source = GCLK_GENERATOR_1;  
    config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV64;  
    config_tcc.counter.period = 2000;  
    config_tcc.compare.match[0] = 900;  
    config_tcc.compare.match[1] = 930;  
    config_tcc.compare.match[2] = 1100;  
    config_tcc.compare.match[3] = 1250;  
  
    tcc_init(&tcc_instance, TCC0, &config_tcc);  
  
    tcc_enable(&tcc_instance);  
}  
  
static void configure_tcc_callbacks(void)  
{  
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,  
        TCC_CALLBACK_OVERFLOW);  
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,  
        TCC_CALLBACK_CHANNEL_0);  
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,  
        TCC_CALLBACK_CHANNEL_1);  
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,  
        TCC_CALLBACK_CHANNEL_2);  
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,  
        TCC_CALLBACK_CHANNEL_3);  
  
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_OVERFLOW);  
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_0);  
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_1);  
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_2);  
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_3);  
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_tcc();  
configure_tcc_callbacks();
```

Workflow

1. Create a module software instance structure for the TCC module to store the TCC driver state while it is in use.

```
struct tcc_module tcc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TCC module.
 1. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```
struct tcc_config config_tcc;
```

2. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, TCC0);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TCC settings to configure the GCLK source, prescaler, period, and compare channel values.

```
config_tcc.counter.clock_source = GCLK_GENERATOR_1;
config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV64;
config_tcc.counter.period = 2000;
config_tcc.compare.match[0] = 900;
config_tcc.compare.match[1] = 930;
config_tcc.compare.match[2] = 1100;
config_tcc.compare.match[3] = 1250;
```

4. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, TCC0, &config_tcc);
```

5. Enable the TCC module to start the timer.

```
tcc_enable(&tcc_instance);
```

3. Configure the TCC callbacks.

1. Register the Overflow and Compare Channel Match callback functions with the driver.

```
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
    TCC_CALLBACK_OVERFLOW);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
    TCC_CALLBACK_CHANNEL_0);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
    TCC_CALLBACK_CHANNEL_1);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
    TCC_CALLBACK_CHANNEL_2);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
    TCC_CALLBACK_CHANNEL_3);
```

2. Enable the Overflow and Compare Channel Match callbacks so that it will be called by the driver when appropriate.

```
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_OVERFLOW);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_0);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_1);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_2);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_3);
```

26.8.3.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();  
  
while (true) {  
}
```

Workflow

1. Enter an infinite loop while the timer is running.

```
while (true) {  
}
```

26.8.4. Quick Start Guide for TCC - Callback

The supported board list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. When the PWM signal connects to LED, LED will light. To see the waveform, you may need an oscilloscope.

The PWM output is set up as follows:

Board	Pin	Connect to
SAM D21 Xpro	PB30	LED0
SAM R21 Xpro	PA19	LED0
SAM L21 Xpro	PB10	LED0
SAM L22 Xpro	PC27	LED0
SAM DA1 Xpro	PB30	LED0
SAM C21 Xpro	PA15	LED0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No faults or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations

- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0

26.8.4.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

```
#if SAMR30

#define CONF_PWM_MODULE      LED_0_PWM3CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM3CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM3CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM3CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM3CTRL_MUX
#else

#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
#endif
```

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

Copy-paste the following callback function code to your user application:

```
static void tcc_callback_to_change_duty_cycle(
    struct tcc_module *const module_inst)
{
    static uint32_t delay = 10;
    static uint32_t i = 0;

    if (--delay) {
        return;
    }
    delay = 10;
    i = (i + 0x0800) & 0xFFFF;
    tcc_set_compare_value(module_inst,
        (enum tcc_match_capture_channel)
            (TCC_MATCH_CAPTURE_CHANNEL_0 + CONF_PWM_CHANNEL),
        i + 1);
}
```

Copy-paste the following setup code to your user application:

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
    tcc_enable(&tcc_instance);
}

static void configure_tcc_callbacks(void)
{
    tcc_register_callback(
        &tcc_instance,
        tcc_callback_to_change_duty_cycle,
        (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));

    tcc_enable_callback(&tcc_instance,
        (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_tcc();
configure_tcc_callbacks();
```

Workflow

1. Create a module software instance structure for the TCC module to store the TCC driver state while it is in use.

```
struct tcc_module tcc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the TCC module.
 1. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```
struct tcc_config config_tcc;
```

2. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Alter the TCC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
```

4. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;
```

5. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

6. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

3. Configure the TCC callbacks.

1. Register the Compare Channel 0 Match callback functions with the driver.

```
tcc_register_callback(
    &tcc_instance,
    tcc_callback_to_change_duty_cycle,
    (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 +
CONF_PWM_CHANNEL));
```

2. Enable the Compare Channel 0 Match callback so that it will be called by the driver when appropriate.

```
tcc_enable_callback(&tcc_instance,
    (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 +
CONF_PWM_CHANNEL));
```

26.8.4.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TCC module.

```
while (true) {
```

26.8.5. Quick Start Guide for TCC - Non-Recoverable Fault

The supported kit list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. There is a non-recoverable fault input which controls PWM output. When this fault is active (low) the PWM output will be forced to be high. When fault is released (input high) the PWM output will go on.

When the PWM signal connects to LED, LED will light. If fault input is from a button, the LED will be off when the button is down and on when the button is up. To see the PWM waveform, you may need an oscilloscope.

The PWM output and fault input is set up as follows:

Board	Pin	Connect to
SAM D21 Xpro	PB30	LED0
SAM D21 Xpro	PA15	SW0
SAM R21 Xpro	PA19	LED0
SAM R21 Xpro	PA28	SW0
SAM L21 Xpro	PB10	LED0
SAM L21 Xpro	PA16	SW0
SAM L22 Xpro	PC27	LED0
SAM L22 Xpro	PC01	SW0
SAM DA1 Xpro	PB30	LED0
SAM DA1 Xpro	PA15	SW0
SAM C21 Xpro	PA15	LED0
SAM C21 Xpro	PA28	SW0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No waveform extentions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input except TCC event0 enabled
- No event action except TCC event0 acts as Non-Recoverable Fault
- No event generation enabled

- Counter starts on 0

26.8.5.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN   SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE  SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15
#define CONF_FAULT_EVENT_USER  EVSYS_ID_USER_TCC0_EV_0
```

- SAM R21 Xplained Pro

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN   SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE  SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_8
#define CONF_FAULT_EVENT_USER  EVSYS_ID_USER_TCC0_EV_0
```

- SAM L21 Xplained Pro

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN      LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX      LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN     SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE    SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_2
#define CONF_FAULT_EVENT_USER   EVSYS_ID_USER_TCC0_EV_0
```

- **SAM L22 Xplained Pro**

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN     SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE    SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_9
#define CONF_FAULT_EVENT_USER   EVSYS_ID_USER_TCC0_EV_0
```

- **SAM DA1 Xplained Pro**

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN     SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE    SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15
#define CONF_FAULT_EVENT_USER   EVSYS_ID_USER_TCC0_EV_0
```

- **SAM C21 Xplained Pro**

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
```

```

#define CONF_PWM_CHANNEL      LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN    SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE   SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_8
#define CONF_FAULT_EVENT_USER  EVSYS_ID_USER_TCC0_EV_0

```

Add to the main application source file, before any functions:

```
#include <string.h>
```

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

```
struct events_resource event_resource;
```

Copy-paste the following callback function code to your user application:

```

static void tcc_callback_to_change_duty_cycle(
    struct tcc_module *const module_inst)
{
    static uint32_t delay = 10;
    static uint32_t i = 0;

    if (--delay) {
        return;
    }
    delay = 10;
    i = (i + 0x0800) & 0xFFFF;
    tcc_set_compare_value(module_inst,
        (enum tcc_match_capture_channel)
        (TCC_MATCH_CAPTURE_CHANNEL_0 + CONF_PWM_CHANNEL),
        i + 1);
}

```

```

static void eic_callback_to_clear_halt(void)
{
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
            TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(0));
    }
}

```

Copy-paste the following setup code to your user application:

```

static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
}

```

```

        config_tcc.counter.period = 0xFFFF;
        config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
        config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
        config_tcc.wave_ext.non_recoverable_fault[0].output =
TCC_FAULT_STATE_OUTPUT_1;
        config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
        config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
        config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;

        tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

        struct tcc_events events;
        memset(&events, 0, sizeof(struct tcc_events));

        events.on_input_event_perform_action[0] = true;
        events.input_config[0].modify_action = true;
        events.input_config[0].action =
TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT;

        tcc_enable_events(&tcc_instance, &events);

        tcc_enable(&tcc_instance);
    }

    static void configure_tcc_callbacks(void)
    {
        tcc_register_callback(
            &tcc_instance,
            tcc_callback_to_change_duty_cycle,
            (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 +
CONF_PWM_CHANNEL));

        tcc_enable_callback(&tcc_instance,
            (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 +
CONF_PWM_CHANNEL));
    }

```

```

    static void configure_eic(void)
    {
        struct extint_chan_conf config;
        extint_chan_get_config_defaults(&config);
        config.filter_input_signal = true;
        config.detection_criteria = EXTINT_DETECT_BOTH;
        config.gpio_pin = CONF_FAULT_EIC_PIN;
        config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
        extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);

        struct extint_events events;
        memset(&events, 0, sizeof(struct extint_events));
        events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
        extint_enable_events(&events);

        extint_register_callback(eic_callback_to_clear_halt,
            CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
        extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
            EXTINT_CALLBACK_TYPE_DETECT);
    }

```

```

    static void configure_event(void)
    {

```

```

struct events_config config;

events_get_config_defaults(&config);

config.generator = CONF_FAULT_EVENT_GENERATOR;
config.path      = EVENTS_PATH_ASYNCHRONOUS;

events_allocate(&event_resource, &config);

events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_tcc();
configure_tcc_callbacks();

configure_eic();
configure_event();

```

Workflow

Configure TCC

1. Create a module software instance struct for the TCC module to store the TCC driver state while it is in use.

```

struct tcc_module tcc_instance;

```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```

struct tcc_config config_tcc;

```

3. Initialize the TCC configuration struct with the module's default values.

```

tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Alter the TCC settings to configure the counter width, wave generation mode, and the compare channel 0 value and fault options. Here the Non-Recoverable Fault output is enabled and set to high level (1).

```

config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;

```

```

config_tcc.wave_ext.non_recoverable_fault[0].output =
TCC_FAULT_STATE_OUTPUT_1;

```

5. Alter the TCC settings to configure the PWM output on a physical device pin.

```

config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;

```

6. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

7. Create a TCC events configuration struct, which can be filled out to enable/disable events and configure event settings. Reset all fields to zero.

```
struct tcc_events events;  
memset(&events, 0, sizeof(struct tcc_events));
```

8. Alter the TCC events settings to enable/disable desired events, to change event generating options and modify event actions. Here TCC event0 will act as Non-Recoverable Fault input.

```
events.on_input_event_perform_action[0] = true;  
events.input_config[0].modify_action = true;  
events.input_config[0].action = TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT;
```

9. Enable and apply events settings.

```
tcc_enable_events(&tcc_instance, &events);
```

10. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

11. Register the Compare Channel 0 Match callback functions with the driver.

```
tcc_register_callback(  
    &tcc_instance,  
    tcc_callback_to_change_duty_cycle,  
    (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

12. Enable the Compare Channel 0 Match callback so that it will be called by the driver when appropriate.

```
tcc_enable_callback(&tcc_instance,  
    (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

Configure EXTINT for fault input

1. Create an EXTINT module channel configuration struct, which can be filled out to adjust the configuration of a single external interrupt channel.

```
struct extint_chan_conf config;
```

2. Initialize the channel configuration struct with the module's default values.

```
extint_chan_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to configure the pin MUX (to route the desired physical pin to the logical channel) to the board button, and to configure the channel to detect both rising and falling edges.

```
config.filter_input_signal = true;  
config.detection_criteria = EXTINT_DETECT_BOTH;  
config.gpio_pin = CONF_FAULT_EIC_PIN;  
config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
```

4. Configure external interrupt channel with the desired channel settings.

```
extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);
```

5. Create a TXTINT events configuration struct, which can be filled out to enable/disable events. Reset all fields to zero.

```
struct extint_events events;  
memset(&events, 0, sizeof(struct extint_events));
```

6. Adjust the configuration struct, set the channels to be enabled to `true`. Here the channel to the board button is used.

```
events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
```

7. Enable the events.

```
extint_enable_events(&events);
```

8. Define the EXTINT callback that will be fired when a detection event occurs. For this example, when fault line is released, the TCC fault state is cleared to go on PWM generating.

```
static void eic_callback_to_clear_halt(void)  
{  
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {  
        tcc_clear_status(&tcc_instance,  
                        TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(0));  
    }  
}
```

9. Register a callback function `eic_callback_to_clear_halt()` to handle detections from the External Interrupt Controller (EIC).

```
extint_register_callback(eic_callback_to_clear_halt,  
                        CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
```

10. Enable the registered callback function for the configured External Interrupt channel, so that it will be called by the module when the channel detects an edge.

```
extint_chan_enable_callback(CONF_FAULT_EIC_LINE,  
                           EXTINT_CALLBACK_TYPE_DETECT);
```

Configure EVENTS for fault input

1. Create an event resource instance struct for the EVENTS module to store.

```
struct events_resource event_resource;
```

Note: This should never go out of scope as long as the resource is in use. In most cases, this should be global.

2. Create an event channel configuration struct, which can be filled out to adjust the configuration of a single event channel.

```
struct events_config config;
```

3. Initialize the event channel configuration struct with the module's default values.

```
events_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

- Adjust the configuration struct to request that the channel will be attached to the specified event generator, and that the asynchronous event path will be used. Here the EIC channel connected to board button is the event generator.

```
config.generator = CONF_FAULT_EVENT_GENERATOR;
config.path       = EVENTS_PATH_ASYNCHRONOUS;
```

- Allocate and configure the channel using the configuration structure.

```
events_allocate(&event_resource, &config);
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

- Attach a user to the channel. Here the user is TCC event0, which has been configured as input of Non-Recoverable Fault.

```
events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
```

26.8.5.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

Workflow

- Enter an infinite loop while the PWM wave is generated via the TCC module.

```
while (true) {
}
```

26.8.6. Quick Start Guide for TCC - Recoverable Fault

The supported board list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. There is a recoverable fault input which controls PWM output. When this fault is active (low) the PWM output will be frozen (could be off or on, no light changing). When fault is released (input high) the PWM output will go on.

When the PWM signal connects to LED, LED will light. If fault input is from a button, the LED will be frozen and not changing its light when the button is down and will go on when the button is up. To see the PWM waveform, you may need an oscilloscope.

The PWM output and fault input is set up as follows:

Board	Pin	Connect to
SAM D21 Xpro	PB30	LED0
SAM D21 Xpro	PA15	SW0
SAM R21 Xpro	PA06	EXT1 Pin 3

Board	Pin	Connect to
SAM R21 Xpro	PA28	SW0
SAM L21 Xpro	PB10	LED0
SAM L21 Xpro	PA16	SW0
SAM L22 Xpro	PB18	EXT3 Pin 9
SAM L22 Xpro	PC01	SW0
SAM DA1 Xpro	PB30	LED0
SAM DA1 Xpro	PA15	SW0
SAM C21 Xpro	PA15	LED0
SAM C21 Xpro	PA28	SW0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No waveform extentions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input except channel 0 event enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Recoverable Fault A is generated from channel 0 event input, fault halt acts as software halt, other actions or options are all disabled

26.8.6.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D21 Xplained Pro

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL    LED_0_PWM4CTRL_CHANNEL
```

```

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN    SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE   SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15
#define CONF_FAULT_EVENT_USER   EVSYS_ID_USER_TCC0_MC_0

```

- **SAM R21 Xplained Pro**

```

#define CONF_PWM_MODULE      TCC1
#define CONF_PWM_CHANNEL     0
#define CONF_PWM_OUTPUT      0
#define CONF_PWM_OUT_PIN     PIN_PA06E_TCC1_WO0
#define CONF_PWM_OUT_MUX     MUX_PA06E_TCC1_WO0

#define CONF_FAULT_EIC_PIN    SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE   SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_8
#define CONF_FAULT_EVENT_USER   EVSYS_ID_USER_TCC1_MC_0

```

- **SAM L21 Xplained Pro**

```

#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX

#define CONF_FAULT_EIC_PIN    SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE   SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_2
#define CONF_FAULT_EVENT_USER   EVSYS_ID_USER_TCC0_MC_0

```

- **SAM L22 Xplained Pro**

```
#define CONF_PWM_MODULE          TCC0

#define CONF_PWM_CHANNEL         0

#define CONF_PWM_OUTPUT          0

#define CONF_PWM_OUT_PIN         PIN_PB18F_TCC0_WO0

#define CONF_PWM_OUT_MUX         MUX_PB18F_TCC0_WO0


#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN

#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX

#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE

#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_9

#define CONF_FAULT_EVENT_USER    EVSYS_ID_USER_TCC0_MC_0
```

- **SAM DA1 Xplained Pro**

```
#define CONF_PWM_MODULE          LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL         LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT          LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN         LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX         LED_0_PWM4CTRL_MUX


#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN

#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX

#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE

#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15

#define CONF_FAULT_EVENT_USER    EVSYS_ID_USER_TCC0_MC_0
```

- **SAM C21 Xplained Pro**

```
#define CONF_PWM_MODULE          LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL         LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT          LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN         LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX         LED_0_PWM4CTRL_MUX


#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN

#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX

#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE
```

```
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_8

#define CONF_FAULT_EVENT_USER      EVSYS_ID_USER_TCC0_EV_0
```

Add to the main application source file, before any functions:

```
#include <string.h>
```

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

```
struct events_resource event_resource;
```

Copy-paste the following callback function code to your user application:

```
static void tcc_callback_to_change_duty_cycle(
    struct tcc_module *const module_inst)
{
    static uint32_t delay = 10;
    static uint32_t i = 0;

    if (--delay) {
        return;
    }
    delay = 10;
    i = (i + 0x0800) & 0xFFFF;
    tcc_set_compare_value(module_inst,
        (enum tcc_match_capture_channel)
        (TCC_MATCH_CAPTURE_CHANNEL_0 + CONF_PWM_CHANNEL),
        i + 1);
}

static void eic_callback_to_clear_halt(void)
{
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
            TCC_STATUS_RECOVERABLE_FAULT_OCCUR(CONF_PWM_CHANNEL));
    }
}
```

Copy-paste the following setup code to your user application:

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
    config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].source =
        TCC_FAULT_SOURCE_ENABLE;

    config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].halt_action =
        TCC_FAULT_HALT_ACTION_SW_HALT;
    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;
}
```

```

tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

struct tcc_events events;
memset(&events, 0, sizeof(struct tcc_events));

events.on_event_perform_channel_action[CONF_PWM_CHANNEL] = true;

tcc_enable_events(&tcc_instance, &events);

tcc_enable(&tcc_instance);
}

static void configure_tcc_callbacks(void)
{
    tcc_register_callback(
        &tcc_instance,
        tcc_callback_to_change_duty_cycle,
        (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 +
CONF_PWM_CHANNEL));

    tcc_enable_callback(&tcc_instance,
        (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 +
CONF_PWM_CHANNEL));
}

```

```

static void configure_eic(void)
{
    struct extint_chan_conf config;
    extint_chan_get_config_defaults(&config);
    config.filter_input_signal = true;
    config.detection_criteria = EXTINT_DETECT_BOTH;
    config.gpio_pin = CONF_FAULT_EIC_PIN;
    config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
    extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);

    struct extint_events events;
    memset(&events, 0, sizeof(struct extint_events));
    events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
    extint_enable_events(&events);

    extint_register_callback(eic_callback_to_clear_halt,
        CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
    extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
        EXTINT_CALLBACK_TYPE_DETECT);
}

```

```

static void configure_event(void)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator = CONF_FAULT_EVENT_GENERATOR;
    config.path = EVENTS_PATH_ASYNCHRONOUS;

    events_allocate(&event_resource, &config);

    events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
}

```

Add to user application initialization (typically the start of `main()`):

```

configure_tcc();
configure_tcc_callbacks();

```

```
configure_eic();
configure_event();
```

Workflow

Configure TCC

1. Create a module software instance struct for the TCC module to store the TCC driver state while it is in use.

```
struct tcc_module tcc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```
struct tcc_config config_tcc;
```

3. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Alter the TCC settings to configure the counter width, wave generation mode, and the compare channel 0 value and fault options. Here the Recoverable Fault input is enabled and halt action is set to software mode (must use software to clear halt state).

```
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
```

```
config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].source =
TCC_FAULT_SOURCE_ENABLE;
config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].halt_action =
TCC_FAULT_HALT_ACTION_SW_HALT;
```

5. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;
```

6. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

7. Create a TCC events configuration struct, which can be filled out to enable/disable events and configure event settings. Reset all fields to zero.

```
struct tcc_events events;
memset(&events, 0, sizeof(struct tcc_events));
```

8. Alter the TCC events settings to enable/disable desired events, to change event generating options and modify event actions. Here channel event 0 input is enabled as source of recoverable fault.

```
events.on_event_perform_channel_action[CONF_PWM_CHANNEL] = true;
```

9. Enable and apply events settings.

```
tcc_enable_events(&tcc_instance, &events);
```

10. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

11. Register the Compare Channel 0 Match callback functions with the driver.

```
tcc_register_callback(  
    &tcc_instance,  
    tcc_callback_to_change_duty_cycle,  
    (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

12. Enable the Compare Channel 0 Match callback so that it will be called by the driver when appropriate.

```
tcc_enable_callback(&tcc_instance,  
    (enum tcc_callback) (TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

Configure EXTINT for fault input

1. Create an EXTINT module channel configuration struct, which can be filled out to adjust the configuration of a single external interrupt channel.

```
struct extint_chan_conf config;
```

2. Initialize the channel configuration struct with the module's default values.

```
extint_chan_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to configure the pin MUX (to route the desired physical pin to the logical channel) to the board button, and to configure the channel to detect both rising and falling edges.

```
config.filter_input_signal = true;  
config.detection_criteria = EXTINT_DETECT_BOTH;  
config.gpio_pin = CONF_FAULT_EIC_PIN;  
config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
```

4. Configure external interrupt channel with the desired channel settings.

```
extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);
```

5. Create a EXTINT events configuration struct, which can be filled out to enable/disable events. Reset all fields to zero.

```
struct extint_events events;  
memset(&events, 0, sizeof(struct extint_events));
```

6. Adjust the configuration struct, set the channels to be enabled to true. Here the channel to the board button is used.

```
events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
```

7. Enable the events.

```
extint_enable_events(&events);
```


8. Define the EXTINT callback that will be fired when a detection event occurs. For this example, when fault line is released, the TCC fault state is cleared to go on PWM generating.

```
static void eic_callback_to_clear_halt(void)
{
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
            TCC_STATUS_RECOVERABLE_FAULT_OCCUR(CONF_PWM_CHANNEL));
    }
}
```

9. Register a callback function `eic_callback_to_clear_halt()` to handle detections from the External Interrupt Controller (EIC).

```
extint_register_callback(eic_callback_to_clear_halt,
    CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
```

10. Enable the registered callback function for the configured External Interrupt channel, so that it will be called by the module when the channel detects an edge.

```
extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
    EXTINT_CALLBACK_TYPE_DETECT);
```

Configure EVENTS for fault input

1. Create an event resource instance struct for the EVENTS module to store.

```
struct events_resource event_resource;
```

Note: This should never go out of scope as long as the resource is in use. In most cases, this should be global.

2. Create an event channel configuration struct, which can be filled out to adjust the configuration of a single event channel.

```
struct events_config config;
```

3. Initialize the event channel configuration struct with the module's default values.

```
events_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Adjust the configuration struct to request that the channel will be attached to the specified event generator, and that the asynchronous event path will be used. Here the EIC channel connected to board button is the event generator.

```
config.generator = CONF_FAULT_EVENT_GENERATOR;
config.path      = EVENTS_PATH_ASYNCHRONOUS;
```

5. Allocate and configure the channel using the configuration structure.

```
events_allocate(&event_resource, &config);
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

6. Attach a user to the channel. Here the user is TCC channel 0 event, which has been configured as input of Recoverable Fault.

```
events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
```

26.8.6.2. Use Case

Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();  
  
while (true) {  
}
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TCC module.

```
while (true) {  
}
```

26.8.7. Quick Start Guide for Using DMA with TCC

The supported board list:

- SAM D21/R21/L21/L22/DA1/C21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal. Here the pulse width varies through the following values with the help of DMA transfer: one quarter of the period, half of the period, and three quarters of the period. The PWM output can be used to drive a LED. The waveform can also be viewed using an oscilloscope. The output signal is also fed back to another TCC channel by event system, the event stamps are captured and transferred to a buffer by DMA.

The PWM output is set up as follows:

Board	Pin	Connect to
SAM D21 Xpro	PB30	LED0
SAM R21 Xpro	PA19	LED0
SAM L21 Xpro	PB10	LED0
SAM L22 Xpro	PC27	LED0
SAM DA1 Xpro	PB30	LED0
SAM C21 Xpro	PA15	LED0

The TCC module will be setup as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- Prescaler is set to 1024
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No fault or waveform extensions
- No inversion of waveform output
- No capture enabled

- Count upward
- Don't perform one-shot operations
- Counter starts on 0
- Counter top set to 0x1000
- Channel 0 (on SAM D21 Xpro) or 3 (on SAM R21 Xpro) is set to compare and match value 0x1000*3/4 and generate event
- Channel 1 is set to capture on input event

The event resource of EVSYS module will be setup as follows:

- TCC match capture channel 0 (on SAM D21 Xpro) or 3 (on SAM R21 Xpro) is selected as event generator
- Event generation is synchronous, with rising edge detected
- TCC match capture channel 1 is the event user

The DMA resource of DMAC module will be setup as follows:

- Two DMA resources are used
- Both DMA resources use peripheral trigger
- Both DMA resources perform beat transfer on trigger
- Both DMA resources use beat size of 16 bits
- Both DMA resources are configured to transfer three beats and then repeat again in same buffer
- On DMA resource which controls the compare value
 - TCC0 overflow triggers DMA transfer
 - The source address increment is enabled
 - The destination address is fixed to TCC channel 0 Compare/Capture register
- On DMA resource which reads the captured value
 - TCC0 capture on channel 1 triggers DMA transfer
 - The source address is fixed to TCC channel 1 Compare/Capture register
 - The destination address increment is enabled
 - The captured value is transferred to an array in SRAM

26.8.7.1. Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D21 Xplained Pro

```
#define CONF_PWM_MODULE          LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL        LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT         LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN        LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX        LED_0_PWM4CTRL_MUX

#define CONF_TCC_CAPTURE_CHANNEL 1

#define CONF_TCC_EVENT_GENERATOR EVSYS_ID_GEN_TCC0_MCX_0
```

```
#define CONF_TCC_EVENT_USER          EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

```
#define CONF_CAPTURE_TRIGGER TCC0_DMAC_ID_MC_1
```

- **SAM R21 Xplained Pro**

```
#define CONF_PWM_MODULE          LED_0_PWM4CTRL_MODULE
```

```
#define CONF_PWM_CHANNEL        LED_0_PWM4CTRL_CHANNEL
```

```
#define CONF_PWM_OUTPUT         LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN        LED_0_PWM4CTRL_PIN
```

```
#define CONF_PWM_OUT_MUX        LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL 1
```

```
#define CONF_TCC_EVENT_GENERATOR EVSYS_ID_GEN_TCC0_MCX_3
```

```
#define CONF_TCC_EVENT_USER      EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

```
#define CONF_CAPTURE_TRIGGER TCC0_DMAC_ID_MC_1
```

- **SAM L21 Xplained Pro**

```
#define CONF_PWM_MODULE          LED_0_PWM4CTRL_MODULE
```

```
#define CONF_PWM_CHANNEL        LED_0_PWM4CTRL_CHANNEL
```

```
#define CONF_PWM_OUTPUT         LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN        LED_0_PWM4CTRL_PIN
```

```
#define CONF_PWM_OUT_MUX        LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL 1
```

```
#define CONF_TCC_EVENT_GENERATOR EVSYS_ID_GEN_TCC0_MCX_0
```

```
#define CONF_TCC_EVENT_USER      EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

- **SAM L22 Xplained Pro**

```
#define CONF_PWM_MODULE          LED_0_PWM4CTRL_MODULE
```

```
#define CONF_PWM_CHANNEL        LED_0_PWM4CTRL_CHANNEL
```

```
#define CONF_PWM_OUTPUT         LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN        LED_0_PWM4CTRL_PIN
```

```
#define CONF_PWM_OUT_MUX      LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL      1
```

```
#define CONF_TCC_EVENT_GENERATOR      EVSYS_ID_GEN_TCC0_MCX_0
```

```
#define CONF_TCC_EVENT_USER      EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

- **SAM DA1 Xplained Pro**

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
```

```
#define CONF_PWM_CHANNEL      LED_0_PWM4CTRL_CHANNEL
```

```
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN      LED_0_PWM4CTRL_PIN
```

```
#define CONF_PWM_OUT_MUX      LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL      1
```

```
#define CONF_TCC_EVENT_GENERATOR      EVSYS_ID_GEN_TCC0_MCX_0
```

```
#define CONF_TCC_EVENT_USER      EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

```
#define CONF_CAPTURE_TRIGGER TCC0_DMAC_ID_MC_1
```

- **SAM C21 Xplained Pro**

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
```

```
#define CONF_PWM_CHANNEL      LED_0_PWM4CTRL_CHANNEL
```

```
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN      LED_0_PWM4CTRL_PIN
```

```
#define CONF_PWM_OUT_MUX      LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL      1
```

```
#define CONF_TCC_EVENT_GENERATOR      EVSYS_ID_GEN_TCC0_MCX_0
```

```
#define CONF_TCC_EVENT_USER      EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

Add to the main application source file, outside of any functions:

```
struct tcc_module tcc_instance;
```

```
uint16_t capture_values[3] = {0, 0, 0};
```

```
struct dma_resource capture_dma_resource;
```

```
COMPILER_ALIGNED(16) DmacDescriptor capture_dma_descriptor
SECTION_DMAC_DESCRIPTOR;
struct events_resource capture_event_resource;
```

```
uint16_t compare_values[3] = {
    (0x1000 / 4), (0x1000 * 2 / 4), (0x1000 * 3 / 4)
};
struct dma_resource compare_dma_resource;
COMPILER_ALIGNED(16) DmacDescriptor compare_dma_descriptor
SECTION_DMAC_DESCRIPTOR;
```

Copy-paste the following setup code to your user application:

```
static void config_event_for_capture(void)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator      = CONF_TCC_EVENT_GENERATOR;
    config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
    config.path           = EVENTS_PATH_SYNCHRONOUS;
    config.clock_source   = GCLK_GENERATOR_0;

    events_allocate(&capture_event_resource, &config);

    events_attach_user(&capture_event_resource, CONF_TCC_EVENT_USER);
}
```

```
static void config_dma_for_capture(void)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
    config.peripheral_trigger = CONF_CAPTURE_TRIGGER;

    dma_allocate(&capture_dma_resource, &config);

    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = 3;
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.step_selection = DMA_STEPSEL_SRC;
    descriptor_config.src_increment_enable = false;
    descriptor_config.source_address =
        (uint32_t)&CONF_PWM_MODULE->CC[CONF_TCC_CAPTURE_CHANNEL];
    descriptor_config.destination_address =
        (uint32_t)capture_values + sizeof(capture_values);

    dma_descriptor_create(&capture_dma_descriptor, &descriptor_config);

    dma_add_descriptor(&capture_dma_resource, &capture_dma_descriptor);
    dma_add_descriptor(&capture_dma_resource, &capture_dma_descriptor);
    dma_start_transfer_job(&capture_dma_resource);
}
```

```
static void config_dma_for_wave(void)
{
    struct dma_resource_config config;
```

```

    dma_get_config_defaults(&config);
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
    config.peripheral_trigger = CONF_COMPARE_TRIGGER;
    dma_allocate(&compare_dma_resource, &config);

    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = 3;
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.source_address =
        (uint32_t)compare_values + sizeof(compare_values);
#if (SAMR21) || (SAMD21) || (SAMD1)
    descriptor_config.destination_address =
        (uint32_t)&CONF_PWM_MODULE->CC[CONF_PWM_CHANNEL];
#else
    descriptor_config.destination_address =
        (uint32_t)&CONF_PWM_MODULE->CCBUF[CONF_PWM_CHANNEL];
#endif

    dma_descriptor_create(&compare_dma_descriptor, &descriptor_config);

    dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
    dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
    dma_start_transfer_job(&compare_dma_resource);
}

```

```

static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV1024;
    config_tcc.counter.period = 0x1000;
    config_tcc.compare.channel_function[CONF_TCC_CAPTURE_CHANNEL] =
        TCC_CHANNEL_FUNCTION_CAPTURE;
    config_tcc.compare.wave_generation =
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.wave_polarity[CONF_PWM_CHANNEL] =
TCC_WAVE_POLARITY_0;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = compare_values[2];

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT] =
CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    struct tcc_events events_tcc = {
        .input_config[0].modify_action = false,
        .input_config[1].modify_action = false,
        .output_config.modify_generation_selection = false,
        .generate_event_on_channel[CONF_PWM_CHANNEL] = true,
        .on_event_perform_channel_action[CONF_TCC_CAPTURE_CHANNEL] =
true
    };
    tcc_enable_events(&tcc_instance, &events_tcc);

    config_event_for_capture();
}

```

```

    config_dma_for_capture();
    config_dma_for_wave();

    tcc_enable(&tcc_instance);
}

```

Add to user application initialization (typically the start of `main()`):

```
configure_tcc();
```

Workflow

Configure the TCC

1. Create a module software instance structure for the TCC module to store the TCC driver state while it is in use.

```
struct tcc_module tcc_instance;
```

Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a TCC module configuration struct, which can be filled out to adjust the configuration of a physical TCC peripheral.

```
struct tcc_config config_tcc;
```

3. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Alter the TCC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

```

config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV1024;
config_tcc.counter.period = 0x1000;
config_tcc.compare.channel_function[CONF_TCC_CAPTURE_CHANNEL] =
    TCC_CHANNEL_FUNCTION_CAPTURE;
config_tcc.compare.wave_generation =
    TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.wave_polarity[CONF_PWM_CHANNEL] =
    TCC_WAVE_POLARITY_0;
config_tcc.compare.match[CONF_PWM_CHANNEL] = compare_values[2];

```

5. Alter the TCC settings to configure the PWM output on a physical device pin.

```

config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]
    = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]
    = CONF_PWM_OUT_MUX;

```

6. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

7. Configure and enable the desired events for the TCC module.

```

struct tcc_events events_tcc = {
    .input_config[0].modify_action = false,
    .input_config[1].modify_action = false,
    .output_config.modify_generation_selection = false,
    .generate_event_on_channel[CONF_PWM_CHANNEL] = true,
}

```



```

        .on_event_perform_channel_action[CONF_TCC_CAPTURE_CHANNEL] = true
    };
    tcc_enable_events(&tcc_instance, &events_tcc);

```

Configure the Event System

Configure the EVSYS module to wire channel 0 event to channel 1.

1. Create an event resource instance.

```
struct events_resource capture_event_resource;
```

Note: This should never go out of scope as long as the resource is in use. In most cases, this should be global.

2. Create an event resource configuration struct.

```
struct events_config config;
```

3. Initialize the event resource configuration struct with default values.

```
events_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Adjust the event resource configuration to desired values.

```

config.generator      = CONF_TCC_EVENT_GENERATOR;
config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
config.path           = EVENTS_PATH_SYNCHRONOUS;
config.clock_source    = GCLK_GENERATOR_0;

```

5. Allocate and configure the resource using the configuration structure.

```
events_allocate(&capture_event_resource, &config);
```

6. Attach a user to the resource.

```
events_attach_user(&capture_event_resource, CONF_TCC_EVENT_USER);
```

Configure the DMA for Capture TCC Channel 1

Configure the DMAC module to obtain captured value from TCC channel 1.

1. Create a DMA resource instance.

```
struct dma_resource capture_dma_resource;
```

Note: This should never go out of scope as long as the resource is in use. In most cases, this should be global.

2. Create a DMA resource configuration struct.

```
struct dma_resource_config config;
```

3. Initialize the DMA resource configuration struct with default values.

```
dma_get_config_defaults(&config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

4. Adjust the DMA resource configurations.

```

config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
config.peripheral_trigger = CONF_CAPTURE_TRIGGER;

```

5. Allocate a DMA resource with the configurations.

```
dma_allocate(&capture_dma_resource, &config);
```

6. Prepare DMA transfer descriptor.

1. Create a DMA transfer descriptor.

```
COMPILER_ALIGNED(16) DmacDescriptor capture_dma_descriptor  
SECTION_DMAL_DESCRIPTOR;
```

Note: When multiple descriptors are linked, the linked item should never go out of scope before it is loaded (to DMA Write-Back memory section). In most cases, if more than one descriptors are used, they should be global except the very first one.

2. Create a DMA transfer descriptor struct.
3. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config descriptor_config;
```

4. Initialize the DMA transfer descriptor configuration struct with default values.

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

5. Adjust the DMA transfer descriptor configurations.

```
descriptor_config.block_transfer_count = 3;  
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;  
descriptor_config.step_selection = DMA_STEPSEL_SRC;  
descriptor_config.src_increment_enable = false;  
descriptor_config.source_address =  
    (uint32_t)&CONF_PWM_MODULE->CC[CONF_TCC_CAPTURE_CHANNEL];  
descriptor_config.destination_address =  
    (uint32_t)capture_values + sizeof(capture_values);
```

6. Create the DMA transfer descriptor with the given configuration.

```
dma_descriptor_create(&capture_dma_descriptor,  
    &descriptor_config);
```

7. Start DMA transfer job with prepared descriptor.

1. Add the DMA transfer descriptor to the allocated DMA resource.

```
dma_add_descriptor(&capture_dma_resource,  
    &capture_dma_descriptor);  
dma_add_descriptor(&capture_dma_resource,  
    &capture_dma_descriptor);
```

Note: When adding multiple descriptors, the last one added is linked at the end of the descriptor queue. If ringed list is needed, just add the first descriptor again to build the circle.

2. Start the DMA transfer job with the allocated DMA resource and transfer descriptor.

```
dma_start_transfer_job(&capture_dma_resource);
```

Configure the DMA for Compare TCC Channel 0

Configure the DMAC module to update TCC channel 0 compare value. The flow is similar to last DMA configure step for capture.

1. Allocate and configure the DMA resource.

```
struct dma_resource compare_dma_resource;
```

```
struct dma_resource_config config;
dma_get_config_defaults(&config);
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
config.peripheral_trigger = CONF_COMPARE_TRIGGER;
dma_allocate(&compare_dma_resource, &config);
```

2. Prepare DMA transfer descriptor.

```
COMPILER_ALIGNED(16) DmacDescriptor compare_dma_descriptor
SECTION_DMAC_DESCRIPTOR;
```

```
struct dma_descriptor_config descriptor_config;

dma_descriptor_get_config_defaults(&descriptor_config);

descriptor_config.block_transfer_count = 3;
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.source_address =
    (uint32_t)compare_values + sizeof(compare_values);
#if (SAMR21) || (SAMD21) || (SAMD11)
    descriptor_config.destination_address =
        (uint32_t)&CONF_PWM_MODULE->CC[CONF_PWM_CHANNEL];
#else
    descriptor_config.destination_address =
        (uint32_t)&CONF_PWM_MODULE->CCBUF[CONF_PWM_CHANNEL];
#endif

dma_descriptor_create(&compare_dma_descriptor, &descriptor_config);
```

3. Start DMA transfer job with prepared descriptor.

```
dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
dma_start_transfer_job(&compare_dma_resource);
```

4. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

26.8.7.2. Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {
    /* Infinite loop */
}
```

Workflow

1. Enter an infinite loop while the PWM wave is generated via the TCC module.

```
while (true) {
    /* Infinite loop */
}
```

27. SAM True Random Number Generator (TRNG) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's True Random Number Generator functionality.

The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- TRNG (True Random Number Generator)

The following devices can use this module:

- Atmel | SMART SAM L21/L22

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

27.1. Prerequisites

There are no prerequisites for this module.

27.2. Module Overview

This driver provides an interface for the TRNG functions on the device.

As soon as the TRNG is enabled, the module provides a new 32-bit random data, for every 84 CLK_TRNG_APB clock cycles.

27.3. Special Considerations

There are no special considerations for this module.

27.4. Extra Information

For extra information, see [Extra Information for TRNG Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

27.5. Examples

For a list of examples related to this driver, see [Examples for TRNG Driver](#).

27.6. API Overview

27.6.1. Variable and Type Definitions

27.6.1.1. Type `trng_callback_t`

```
typedef void(* trng_callback_t )(struct trng_module *const module_inst)
```

Type definition for a TRNG module callback function.

27.6.1.2. Variable `_trng_instance`

```
struct trng_module * _trng_instance
```

27.6.2. Structure Definitions

27.6.2.1. Struct `trng_config`

Configuration structure for a True Random Number Generator.

Table 27-1. Members

Type	Name	Description
bool	run_in_standby	If <code>true</code> , the True Random Number Generator will not be stopped in standby sleep mode

27.6.2.2. Struct `trng_events`

Event flags for the TRNG module. This is used to enable and disable events via [trng_enable_events\(\)](#) and [trng_disable_events\(\)](#).

Table 27-2. Members

Type	Name	Description
bool	generate_event_on_data_ready	Enable event generation on random data ready

27.6.2.3. Struct `trng_module`

TRNG software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

27.6.3. Function Definitions

27.6.3.1. Driver Initialization and Configuration

Function `trng_init()`

Initializes a hardware TRNG module instance.

```
enum status_code trng_init(  
    struct trng_module *const module_inst,  
    Trng *const hw,  
    struct trng_config *const config)
```

Enables the clock and initializes the TRNG module, based on the given configuration values.

Table 27-3. Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the software module instance struct
[in]	hw	Pointer to the TRNG hardware module
[in]	config	Pointer to the TRNG configuration options struct

Returns

Status of the initialization procedure.

Table 27-4. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully

Function `trng_get_config_defaults()`

Initializes all members of a TRNG configuration structure to safe defaults.

```
void trng_get_config_defaults(  
    struct trng_config *const config)
```

Initializes all members of a given True Random Number Generator configuration structure to safe known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- True Random Number Generator will not be stopped in standby sleep mode

Table 27-5. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `trng_enable()`

Enables a TRNG that was previously configured.

```
void trng_enable(  
    struct trng_module *const module_inst)
```

Enables True Random Number Generator that was previously configured via a call to [trng_init\(\)](#).

Table 27-6. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the True Random Number Generator peripheral

Function `trng_disable()`

Disables a TRNG that was previously enabled.

```
void trng_disable(  
    struct trng_module *const module_inst)
```

Disables True Random Number Generator that was previously started via a call to [trng_enable\(\)](#).

Table 27-7. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the True Random Number Generator peripheral

Function `trng_enable_events()`

Enables a TRNG event output.

```
void trng_enable_events(  
    struct trng_module *const module_inst,  
    struct trng_events *const events)
```

Enables output events from the True Random Number Generator module. See Section Struct [trng_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 27-8. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the TRNG peripheral
[in]	events	Struct containing flags of events to enable

Function `trng_disable_events()`

Disables a TRNG event output.

```
void trng_disable_events(  
    struct trng_module *const module_inst,  
    struct trng_events *const events)
```

Disables output events from the True Random Number Generator module. See Section Struct [trng_events](#) for a list of events this module supports.

Note: Events cannot be altered while the module is enabled.

Table 27-9. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the TRNG peripheral
[in]	events	Struct containing flags of events to disable

27.6.3.2. Read TRNG Result

Function `trng_read()`

Read the random data result.

```
enum status_code trng_read(
    struct trng_module *const module_inst,
    uint32_t * result)
```

Reads the random data result.

Table 27-10. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TRNG software instance struct
[out]	result	Pointer to store the result value in

Returns

Status of the TRNG read request.

Table 27-11. Return Values

Return value	Description
STATUS_OK	The result was retrieved successfully
STATUS_BUSY	A random result was not ready

27.6.3.3. Callback Management

Function `trng_register_callback()`

Registers a callback.

```
enum status_code trng_register_callback(
    struct trng_module *const module,
    trng_callback_t callback_func,
    enum trng_callback callback_type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled by [trng_enable_callback](#), in order for the interrupt handler to call it when the conditions for the callback type is met.

Table 27-12. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to TC software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

Table 27-13. Return Values

Return value	Description
STATUS_OK	The function exited successfully

Function trng_unregister_callback()

Unregisters a callback.

```
enum status_code trng_unregister_callback(
    struct trng_module * module,
    enum trng_callback callback_type)
```

Unregisters a callback function implemented by the user. The callback should be disabled before it is unregistered.

Table 27-14. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to TC software instance struct
[in]	callback_type	Callback type given by an enum

Table 27-15. Return Values

Return value	Description
STATUS_OK	The function exited successfully

Function trng_enable_callback()

Enables callback.

```
void trng_enable_callback(
    struct trng_module *const module,
    enum trng_callback callback_type)
```

Enables the callback function registered by [trng_register_callback](#). The callback function will be called from the interrupt handler when the conditions for the callback type are met.

Table 27-16. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to TRNG software instance struct
[in]	callback_type	Callback type given by an enum

Function `trng_disable_callback()`

Disables callback.

```
void trng_disable_callback(  
    struct trng_module *const module,  
    enum trng_callback callback_type)
```

Disables the callback function registered by the `trng_register_callback`.

Table 27-17. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to TRNG software instance struct
[in]	callback_type	Callback type given by an enum

27.6.3.4. Job Management

Function `trng_read_buffer_job()`

Read multiple random data from TRNG.

```
enum status_code trng_read_buffer_job(  
    struct trng_module *const module_inst,  
    uint32_t *buffer,  
    uint32_t number)
```

As soon as the TRNG is enabled, the module provides a new 32-bits random data for every 84 CLK_TRNG_APB clock cycles.

Table 27-18. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TRNG software instance struct
[in]	number	Number of random data to get
[out]	buffer	Buffer to store the random data

Returns

Status of the job start.

Table 27-19. Return Values

Return value	Description
STATUS_OK	The read job was started successfully and is in progress
STATUS_BUSY	The TRNG is already busy with another job

Function `trng_get_job_status()`

Gets the status of a job.

```
enum status_code trng_get_job_status(  
    struct trng_module * module_inst,  
    enum trng_job_type type)
```

Gets the status of an ongoing or the last job.

Table 27-20. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TRNG software instance struct
[in]	type	Type of job to abort

Returns

Status of the job.

Function `trng_abort_job()`

Aborts an ongoing job.

```
void trng_abort_job(  
    struct trng_module * module_inst,  
    enum trng_job_type type)
```

Table 27-21. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TRNG software instance struct
[in]	type	Type of job to abort

27.6.4. Enumeration Definitions

27.6.4.1. Enum `trng_callback`

Enum for possible callback types for the TRNG module.

Table 27-22. Members

Enum value	Description
TRNG_CALLBACK_READ_BUFFER	Callback for specific number of random data ready

27.6.4.2. Enum `trng_job_type`

Enum for the possible types of TRNG asynchronous jobs that may be issued to the driver.

Table 27-23. Members

Enum value	Description
TRNG_JOB_READ_BUFFER	Asynchronous TRNG read into a user provided buffer

27.7. Extra Information for TRNG Driver

27.7.1. Acronyms

Acronym	Description
TRNG	True Random Number Generator

27.7.2. Dependencies

This driver has no dependencies.

27.7.3. Errata

There are no errata related to this driver.

27.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial Release

27.8. Examples for TRNG Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM True Random Number Generator \(TRNG\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that QSGs can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for TRNG - Basic](#)
- [Quick Start Guide for TRNG - Callback](#)

27.8.1. Quick Start Guide for TRNG - Basic

In this use case, the True Random Number Generator (TRNG) module is configured for:

- The TRNG peripheral will not be stopped in standby sleep mode

This use case will read random data in polling mode repeatedly. After reading a data, the board LED will be toggled.

27.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
/* TRNG module software instance (must not go out of scope while in use) */  
static struct trng_module trng_instance;
```

```

void configure_trng(void)
{
    /* Create a new configuration structure for the TRNG settings
    * and fill with the default module settings. */
    struct trng_config config_trng;
    trng_get_config_defaults(&config_trng);

    /* Alter any TRNG configuration settings here if required */

    /* Initialize TRNG with the user settings */
    trng_init(&trng_instance, TRNG, &config_trng);
}

```

Add to user application initialization (typically the start of `main()`):

```

system_init();
configure_trng();
trng_enable(&trng_instance);

```

Workflow

1. Create a TRNG device instance struct, which will be associated with a TRNG peripheral hardware instance.

```
static struct trng_module trng_instance;
```

Note: Device instance structures shall **never** go out of scope when in use.

2. Create a new function `configure_trng()`, which will be used to configure the overall TRNG peripheral.

```
void configure_trng(void)
```

3. Create a TRNG peripheral configuration structure that will be filled out to set the module configuration.

```
struct trng_config config_trng;
```

4. Fill the TRNG peripheral configuration structure with the default module configuration values.

```
trng_get_config_defaults(&config_trng);
```

5. Initialize the TRNG peripheral and associate it with the software instance structure that was defined previously.

```
trng_init(&trng_instance, TRNG, &config_trng);
```

6. Enable the now initialized TRNG peripheral.

```
trng_enable(&trng_instance);
```

27.8.1.2. Implementation

Code

Copy-paste the following code to your user application:

```

uint32_t random_result;

while (true) {
    while (trng_read(&trng_instance, &random_result) != STATUS_OK) {
    }
    port_pin_toggle_output_level(LED_0_PIN);
    /* Add a short delay to see LED toggle */
    volatile uint32_t delay = 50000;
}

```

```

    while(delay--) {
    }
}

```

Workflow

1. Make the application loop infinitely.

```
while (true) {
```

2. Start to read a random data from TRNG until success.

```
while (trng_read(&trng_instance, &random_result) != STATUS_OK) {
}
```

3. Toggle the board LED to indicate a random data is read.

```
port_pin_toggle_output_level(LED_0_PIN);
/* Add a short delay to see LED toggle */
volatile uint32_t delay = 50000;
while(delay--) {
}
```

27.8.2. Quick Start Guide for TRNG - Callback

In this use case, the True Random Number Generator (TRNG) module is configured for:

- The TRNG peripheral will not be stopped in standby sleep mode

This use case will read random data in interrupt mode repeatedly. After reading specific size of buffer data, the board LED will be toggled.

27.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```

bool volatile trng_read_done = false;

void configure_trng(void);
void configure_trng_callback(void);
void trng_complete_callback(struct trng_module *const module_inst);

/* TRNG module software instance (must not go out of scope while in use) */
static struct trng_module trng_instance;

void configure_trng(void)
{
    /* Create a new configuration structure for the TRNG settings
    * and fill with the default module settings. */
    struct trng_config config_trng;
    trng_get_config_defaults(&config_trng);

    /* Alter any TRNG configuration settings here if required */

    /* Initialize TRNG with the user settings */
    trng_init(&trng_instance, TRNG, &config_trng);
}

void trng_complete_callback(struct trng_module *const module_inst)
{

```

```

    trng_read_done = true;
}

void configure_trng_callback(void)
{
    trng_register_callback(&trng_instance, trng_complete_callback,
        TRNG_CALLBACK_READ_BUFFER);
    trng_enable_callback(&trng_instance, TRNG_CALLBACK_READ_BUFFER);
}

```

Add to user application initialization (typically the start of `main()`):

```

system_init();
configure_trng();
configure_trng_callback();

trng_enable(&trng_instance);

```

Workflow

1. Create a TRNG device instance struct, which will be associated with a TRNG peripheral hardware instance.

```
static struct trng_module trng_instance;
```

Note: Device instance structures shall **never** go out of scope when in use.

2. Create a new function `configure_trng()`, which will be used to configure the overall TRNG peripheral.

```
void configure_trng(void)
```

3. Create a TRNG peripheral configuration structure that will be filled out to set the module configuration.

```
struct trng_config config_trng;
```

4. Fill the TRNG peripheral configuration structure with the default module configuration values.

```
trng_get_config_defaults(&config_trng);
```

5. Initialize the TRNG peripheral and associate it with the software instance structure that was defined previously.

```
trng_init(&trng_instance, TRNG, &config_trng);
```

6. Create a new callback function.

```

void trng_complete_callback(struct trng_module *const module_inst)
{
    trng_read_done = true;
}

```

7. Create a callback status software flag.

```
bool volatile trng_read_done = false;
```

8. Let the callback function set the flag to true when read job done.

```
trng_read_done = true;
```

9. Create a new function `configure_trng_callback()`, which will be used to configure the callbacks.

```

void configure_trng_callback(void)
{

```

```

    trng_register_callback(&trng_instance, trng_complete_callback,
                          TRNG_CALLBACK_READ_BUFFER);
    trng_enable_callback(&trng_instance, TRNG_CALLBACK_READ_BUFFER);
}

```

10. Register callback function.

```

trng_register_callback(&trng_instance, trng_complete_callback,
                      TRNG_CALLBACK_READ_BUFFER);

```

11. Enable the callbacks.

```

trng_enable_callback(&trng_instance, TRNG_CALLBACK_READ_BUFFER);

```

12. Enable the now initialized TRNG peripheral.

```

trng_enable(&trng_instance);

```

Note: This should not be done until after the TRNG is set up and ready to be used.

27.8.2.2. Implementation

Code

Copy-paste the following code to your user application:

```

uint32_t random_buffer[5];

while (true) {
    trng_read_buffer_job(&trng_instance, random_buffer, 5);
    while (!trng_read_done) {
    }
    trng_read_done = false;
    port_pin_toggle_output_level(LED_0_PIN);
    /* Add a short delay to see LED toggle */
    volatile uint32_t delay = 50000;
    while(delay--) {
    }
}

```

Workflow

1. Make the application loop infinitely.

```

while (true) {

```

2. Start an asynchronous TRNG read job, to store random data into the global buffer and generate a callback when complete.

```

    trng_read_buffer_job(&trng_instance, random_buffer, 5);

```

3. Wait until the asynchronous read job is complete.

```

    while (!trng_read_done) {
    }
    trng_read_done = false;

```

4. Toggle the board LED to indicate specific size of random data were read.

```

    port_pin_toggle_output_level(LED_0_PIN);
    /* Add a short delay to see LED toggle */
    volatile uint32_t delay = 50000;
    while(delay--) {
    }
}

```


28. SAM Watchdog (WDT) Driver

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Watchdog Timer module, including the enabling, disabling, and kicking within the device. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripherals are used by this module:

- WDT (Watchdog Timer)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

28.1. Prerequisites

There are no prerequisites for this module.

28.2. Module Overview

The Watchdog module (WDT) is designed to give an added level of safety in critical systems, to ensure a system reset is triggered in the case of a deadlock or other software malfunction that prevents normal device operation.

At a basic level, the Watchdog is a system timer with a fixed period; once enabled, it will continue to count ticks of its asynchronous clock until it is periodically reset, or the timeout period is reached. In the event of a Watchdog timeout, the module will trigger a system reset identical to a pulse of the device's reset pin, resetting all peripherals to their power-on default states and restarting the application software from the reset vector.

In many systems, there is an obvious upper bound to the amount of time each iteration of the main application loop can be expected to run, before a malfunction can be assumed (either due to a deadlock waiting on hardware or software, or due to other means). When the Watchdog is configured with a timeout period equal to this upper bound, a malfunction in the system will force a full system reset to allow for a graceful recovery.

28.2.1. Locked Mode

The Watchdog configuration can be set in the device fuses and locked in hardware, so that no software changes can be made to the Watchdog configuration. Additionally, the Watchdog can be locked on in software if it is not already locked, so that the module configuration cannot be modified until a power on reset of the device.

The locked configuration can be used to ensure that faulty software does not cause the Watchdog configuration to be changed, preserving the level of safety given by the module.

28.2.2. Window Mode

Just as there is a reasonable upper bound to the time the main program loop should take for each iteration, there is also in many applications a lower bound, i.e. a *minimum* time for which each loop iteration should run for under normal circumstances. To guard against a system failure resetting the Watchdog in a tight loop (or a failure in the system application causing the main loop to run faster than expected) a "Window" mode can be enabled to disallow resetting of the Watchdog counter before a certain period of time. If the Watchdog is not reset *after* the window opens but not *before* the Watchdog expires, the system will reset.

28.2.3. Early Warning

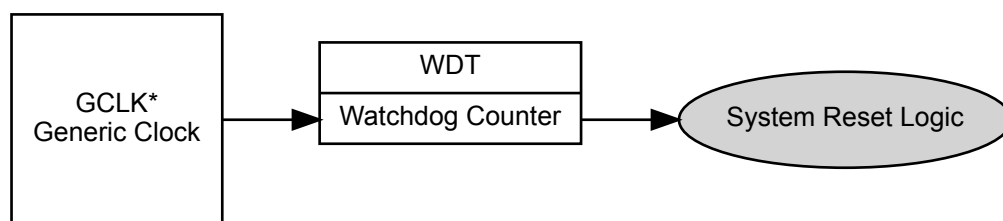
In some cases it is desirable to receive an early warning that the Watchdog is about to expire, so that some system action (such as saving any system configuration data for failure analysis purposes) can be performed before the system reset occurs. The Early Warning feature of the Watchdog module allows such a notification to be requested; after the configured early warning time (but before the expiry of the Watchdog counter) the Early Warning flag will become set, so that the user application can take an appropriate action.

Note: It is important to note that the purpose of the Early Warning feature is *not* to allow the user application to reset the Watchdog; doing so will defeat the safety the module gives to the user application. Instead, this feature should be used purely to perform any tasks that need to be undertaken before the system reset occurs.

28.2.4. Physical Connection

Figure 28-1 shows how this module is interconnected within the device.

Figure 28-1. Physical Connection



Note: Watchdog Counter of SAM L21/L22/R30 is *not* provided by GCLK, but it uses an internal 1KHz OSCULP32K output clock.

28.3. Special Considerations

On some devices the Watchdog configuration can be fused to be always on in a particular configuration; if this mode is enabled the Watchdog is not software configurable and can have its count reset and early warning state checked/cleared only.

28.4. Extra Information

For extra information, see [Extra Information for WDT Driver](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

28.5. Examples

For a list of examples related to this driver, see [Examples for WDT Driver](#).

28.6. API Overview

28.6.1. Variable and Type Definitions

28.6.1.1. Callback Configuration and Initialization

Type `wdt_callback_t`

```
typedef void(* wdt_callback_t )(void)
```

Type definition for a WDT module callback function.

28.6.2. Structure Definitions

28.6.2.1. Struct `wdt_conf`

Configuration structure for a Watchdog Timer instance. This structure should be initialized by the [wdt_get_config_defaults\(\)](#) function before being modified by the user application.

Table 28-1. Members

Type	Name	Description
bool	<code>always_on</code>	If <code>true</code> , the Watchdog will be locked to the current configuration settings when the Watchdog is enabled
enum <code>gclk_generator</code>	<code>clock_source</code>	GCLK generator used to clock the peripheral except SAM L21/L22/C21/C20
enum wdt_period	<code>early_warning_period</code>	Number of Watchdog timer clock ticks until the early warning flag is set
bool	<code>enable</code>	Enable/Disable the Watchdog Timer
enum wdt_period	<code>timeout_period</code>	Number of Watchdog timer clock ticks until the Watchdog expires
enum wdt_period	<code>window_period</code>	Number of Watchdog timer clock ticks until the reset window opens

28.6.3. Function Definitions

28.6.3.1. Configuration and Initialization

Function `wdt_is_syncing()`

Determines if the hardware module(s) are currently synchronizing to the bus.

```
bool wdt_is_syncing( void )
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Returns

Synchronization status of the underlying hardware module(s).

Table 28-2. Return Values

Return value	Description
false	If the module has completed synchronization
true	If the module synchronization is ongoing

Function `wdt_get_config_defaults()`

Initializes a Watchdog Timer configuration structure to defaults.

```
void wdt_get_config_defaults(  
    struct wdt_conf *const config)
```

Initializes a given Watchdog Timer configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Not locked, to allow for further (re-)configuration
- Enable WDT
- Watchdog timer sourced from Generic Clock Channel 4
- A timeout period of 16384 clocks of the Watchdog module clock
- No window period, so that the Watchdog count can be reset at any time
- No early warning period to indicate the Watchdog will soon expire

Table 28-3. Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

Function `wdt_set_config()`

Sets up the WDT hardware module based on the configuration.

```
enum status_code wdt_set_config(  
    const struct wdt_conf *const config)
```

Writes a given configuration of a WDT configuration to the hardware module, and initializes the internal device struct.

Table 28-4. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to the configuration struct

Returns

Status of the configuration procedure.

Table 28-5. Return Values

Return value	Description
STATUS_OK	If the module was configured correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied
STATUS_ERR_IO	If the Watchdog module is locked to be always on

Function `wdt_is_locked()`

Determines if the Watchdog timer is currently locked in an enabled state.

```
bool wdt_is_locked( void )
```

Determines if the Watchdog timer is currently enabled and locked, so that it cannot be disabled or otherwise reconfigured.

Returns

Current Watchdog lock state.

28.6.3.2. Timeout and Early Warning Management

Function `wdt_clear_early_warning()`

Clears the Watchdog timer early warning period elapsed flag.

```
void wdt_clear_early_warning( void )
```

Clears the Watchdog timer early warning period elapsed flag, so that a new early warning period can be detected.

Function `wdt_is_early_warning()`

Determines if the Watchdog timer early warning period has elapsed.

```
bool wdt_is_early_warning( void )
```

Determines if the Watchdog timer early warning period has elapsed.

Note: If no early warning period was configured, the value returned by this function is invalid.

Returns

Current Watchdog Early Warning state.

Function `wdt_reset_count()`

Resets the count of the running Watchdog Timer that was previously enabled.

```
void wdt_reset_count( void )
```

Resets the current count of the Watchdog Timer, restarting the timeout period count elapsed. This function should be called after the window period (if one was set in the module configuration) but before the timeout period to prevent a reset of the system.

28.6.3.3. Callback Configuration and Initialization

Function `wdt_register_callback()`

Registers an asynchronous callback function with the driver.

```
enum status_code wdt_register_callback(  
    const wdt_callback_t callback,  
    const enum wdt_callback type)
```

Registers an asynchronous callback with the WDT driver, fired when a given criteria (such as an Early Warning) is met. Callbacks are fired once for each event.

Table 28-6. Parameters

Data direction	Parameter name	Description
[in]	callback	Pointer to the callback function to register
[in]	type	Type of callback function to register

Returns

Status of the registration operation.

Table 28-7. Return Values

Return value	Description
STATUS_OK	The callback was registered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function `wdt_unregister_callback()`

Unregisters an asynchronous callback function with the driver.

```
enum status_code wdt_unregister_callback(  
    const enum wdt_callback type)
```

Unregisters an asynchronous callback with the WDT driver, removing it from the internal callback registration table.

Table 28-8. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to unregister

Returns

Status of the de-registration operation.

Table 28-9. Return Values

Return value	Description
STATUS_OK	The callback was Unregistered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

28.6.3.4. Callback Enabling and Disabling**Function wdt_enable_callback()**

Enables asynchronous callback generation for a given type.

```
enum status_code wdt_enable_callback(
    const enum wdt_callback type)
```

Enables asynchronous callbacks for a given callback type. This must be called before an external interrupt channel will generate callback events.

Table 28-10. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to enable

Returns

Status of the callback enable operation.

Table 28-11. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

Function wdt_disable_callback()

Disables asynchronous callback generation for a given type.

```
enum status_code wdt_disable_callback(
    const enum wdt_callback type)
```

Disables asynchronous callbacks for a given callback type.

Table 28-12. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to disable

Returns

Status of the callback disable operation.

Table 28-13. Return Values

Return value	Description
STATUS_OK	The callback was disabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

28.6.4. Enumeration Definitions

28.6.4.1. Callback Configuration and Initialization

Enum wdt_callback

Enum for the possible callback types for the WDT module.

Table 28-14. Members

Enum value	Description
WDT_CALLBACK_EARLY_WARNING	Callback type for when an early warning callback from the WDT module is issued

28.6.4.2. Enum wdt_period

Enum for the possible period settings of the Watchdog timer module, for values requiring a period as a number of Watchdog timer clock ticks.

Table 28-15. Members

Enum value	Description
WDT_PERIOD_NONE	No Watchdog period. This value can only be used when setting the Window and Early Warning periods; its use as the Watchdog Reset Period is invalid.
WDT_PERIOD_8CLK	Watchdog period of 8 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_16CLK	Watchdog period of 16 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_32CLK	Watchdog period of 32 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_64CLK	Watchdog period of 64 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_128CLK	Watchdog period of 128 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_256CLK	Watchdog period of 256 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_512CLK	Watchdog period of 512 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_1024CLK	Watchdog period of 1024 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_2048CLK	Watchdog period of 2048 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_4096CLK	Watchdog period of 4096 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_8192CLK	Watchdog period of 8192 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_16384CLK	Watchdog period of 16384 clocks of the Watchdog Timer Generic Clock

28.7. Extra Information for WDT Driver

28.7.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
WDT	Watchdog Timer

28.7.2. Dependencies

This driver has the following dependencies:

- System Clock Driver

28.7.3. Errata

There are no errata related to this driver.

28.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Driver updated to follow driver type convention: • wdt_init, wdt_enable, wdt_disable functions removed • wdt_set_config function added • WDT module enable state moved inside the configuration struct
Initial Release

28.8. Examples for WDT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Watchdog \(WDT\) Driver](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for WDT - Basic](#)
- [Quick Start Guide for WDT - Callback](#)

28.8.1. Quick Start Guide for WDT - Basic

In this use case, the Watchdog module is configured for:

- System reset after 2048 clocks of the Watchdog generic clock
- Always on mode disabled
- Basic mode, with no window or early warning periods

This use case sets up the Watchdog to force a system reset after every 2048 clocks of the Watchdog's Generic Clock channel, unless the user periodically resets the Watchdog counter via a button before the timer expires. If the Watchdog resets the device, a LED on the board is turned off.

28.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_wdt(void)
{
    /* Create a new configuration structure for the Watchdog settings and fill
    * with the default module settings. */
    struct wdt_conf config_wdt;
    wdt_get_config_defaults(&config_wdt);

    /* Set the Watchdog configuration settings */
    config_wdt.always_on = false;
    #if !((SAML21) || (SAMC21) || (SAML22) || (SAMR30))
    config_wdt.clock_source = GCLK_GENERATOR_4;
    #endif
    config_wdt.timeout_period = WDT_PERIOD_2048CLK;

    /* Initialize and enable the Watchdog with the user settings */
    wdt_set_config(&config_wdt);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_wdt();
```

Workflow

1. Create a Watchdog module configuration struct, which can be filled out to adjust the configuration of the Watchdog.

```
struct wdt_conf config_wdt;
```

2. Initialize the Watchdog configuration struct with the module's default values.

```
wdt_get_config_defaults(&config_wdt);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to set the timeout period and lock mode of the Watchdog.

```
config_wdt.always_on = false;
#if !((SAML21) || (SAMC21) || (SAML22) || (SAMR30))
    config_wdt.clock_source = GCLK_GENERATOR_4;
#endif
config_wdt.timeout_period = WDT_PERIOD_2048CLK;
```

4. Setups the WDT hardware module with the requested settings.

```
wdt_set_config(&config_wdt);
```

28.8.1.2. Quick Start Guide for WDT - Basic

Code

Copy-paste the following code to your user application:

```
enum system_reset_cause reset_cause = system_get_reset_cause();

if (reset_cause == SYSTEM_RESET_CAUSE_WDT) {
    port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
}
else {
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
}

while (true) {
    if (port_pin_get_input_level(BUTTON_0_PIN) == false) {
        port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

        wdt_reset_count();
    }
}
```

Workflow

1. Retrieve the cause of the system reset to determine if the Watchdog module was the cause of the last reset.

```
enum system_reset_cause reset_cause = system_get_reset_cause();
```

2. Turn on or off the board LED based on whether the Watchdog reset the device.

```
if (reset_cause == SYSTEM_RESET_CAUSE_WDT) {
    port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
}
else {
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
}
```

3. Enter an infinite loop to hold the main program logic.

```
while (true) {
```

4. Test to see if the board button is currently being pressed.

```
if (port_pin_get_input_level(BUTTON_0_PIN) == false) {
```

5. If the button is pressed, turn on the board LED and reset the Watchdog timer.

```
port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

wdt_reset_count();
```

28.8.2. Quick Start Guide for WDT - Callback

In this use case, the Watchdog module is configured for:

- System reset after 4096 clocks of the Watchdog generic clock
- Always on mode disabled
- Early warning period of 2048 clocks of the Watchdog generic clock

This use case sets up the Watchdog to force a system reset after every 4096 clocks of the Watchdog's Generic Clock channel, with an Early Warning callback being generated every 2048 clocks. Each time the

Early Warning interrupt fires the board LED is turned on, and each time the device resets the board LED is turned off, giving a periodic flashing pattern.

28.8.2.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void watchdog_early_warning_callback(void)
{
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
}

void configure_wdt(void)
{
    /* Create a new configuration structure for the Watchdog settings and fill
     * with the default module settings. */
    struct wdt_conf config_wdt;
    wdt_get_config_defaults(&config_wdt);

    /* Set the Watchdog configuration settings */
    config_wdt.always_on = false;
    #if !((SAML21) || (SAMC21) || (SAML22) || (SAMR30))
    config_wdt.clock_source = GCLK_GENERATOR_4;
    #endif
    config_wdt.timeout_period = WDT_PERIOD_4096CLK;
    config_wdt.early_warning_period = WDT_PERIOD_2048CLK;

    /* Initialize and enable the Watchdog with the user settings */
    wdt_set_config(&config_wdt);
}

void configure_wdt_callbacks(void)
{
    wdt_register_callback(watchdog_early_warning_callback,
        WDT_CALLBACK_EARLY_WARNING);

    wdt_enable_callback(WDT_CALLBACK_EARLY_WARNING);
}
```

Add to user application initialization (typically the start of `main()`):

```
configure_wdt();
configure_wdt_callbacks();
```

Workflow

1. Configure and enable the Watchdog driver.
 1. Create a Watchdog module configuration struct, which can be filled out to adjust the configuration of the Watchdog.

```
struct wdt_conf config_wdt;
```

2. Initialize the Watchdog configuration struct with the module's default values.

```
wdt_get_config_defaults(&config_wdt);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Adjust the configuration struct to set the timeout and early warning periods of the Watchdog.

```
config_wdt.always_on = false;
#if !((SAML21) || (SAMC21) || (SAML22) || (SAMR30))
    config_wdt.clock_source = GCLK_GENERATOR_4;
#endif
    config_wdt.timeout_period = WDT_PERIOD_4096CLK;
config_wdt.early_warning_period = WDT_PERIOD_2048CLK;
```

4. Sets up the WDT hardware module with the requested settings.

```
wdt_set_config(&config_wdt);
```

2. Register and enable the Early Warning callback handler.

1. Register the user-provided Early Warning callback function with the driver, so that it will be run when an Early Warning condition occurs.

```
wdt_register_callback(watchdog_early_warning_callback,
    WDT_CALLBACK_EARLY_WARNING);
```

2. Enable the Early Warning callback so that it will generate callbacks.

```
wdt_enable_callback(WDT_CALLBACK_EARLY_WARNING);
```

28.8.2.2. Quick Start Guide for WDT - Callback

Code

Copy-paste the following code to your user application:

```
port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
system_interrupt_enable_global();

while (true) {
    /* Wait for callback */
}
```

Workflow

1. Turn off the board LED when the application starts.

```
port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
```

2. Enable global interrupts so that callbacks can be generated.

```
system_interrupt_enable_global();
```

3. Enter an infinite loop to hold the main program logic.

```
while (true) {
    /* Wait for callback */
}
```

29. SAM EEPROM Emulator (EEPROM) Service

This driver for Atmel® | SMART ARM®-based microcontrollers provides an emulated EEPROM memory space in the device's FLASH memory, for the storage and retrieval of user-application configuration data into and out of non-volatile memory.

The following peripherals are used by this module:

- NVM (Non-Volatile Memory Controller)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM C20/C21
- Atmel | SMART SAM DA1

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

29.1. Prerequisites

The SAM device fuses must be configured via an external programmer or debugger, so that an EEPROM section is allocated in the main NVM flash memory contents. If a NVM section is not allocated for the EEPROM emulator, or if insufficient space for the emulator is reserved, the module will fail to initialize.

29.2. Module Overview

As the SAM devices do not contain any physical EEPROM memory, the storage of non-volatile user data is instead emulated using a special section of the device's main FLASH memory. The use of FLASH memory technology over EEPROM presents several difficulties over true EEPROM memory; data must be written as a number of physical memory pages (of several bytes each) rather than being individually byte addressable, and entire rows of FLASH must be erased before new data may be stored. To help abstract these characteristics away from the user application an emulation scheme is implemented to present a more user-friendly API for data storage and retrieval.

This module provides an EEPROM emulation layer on top of the device's internal NVM controller, to provide a standard interface for the reading and writing of non-volatile configuration data. This data is placed into the EEPROM emulated section of the device's main FLASH memory storage section, the size of which is configured using the device's fuses. Emulated EEPROM is exempt from the usual device NVM region lock bits, so that it may be read from or written to at any point in the user application.

There are many different algorithms that may be employed for EEPROM emulation using FLASH memory, to tune the write and read latencies, RAM usage, wear levelling and other characteristics. As a

result, multiple different emulator schemes may be implemented, so that the most appropriate scheme for a specific application's requirements may be used.

29.2.1. Implementation Details

The following information is relevant for **EEPROM Emulator scheme 1, version 1.0.0**, as implemented by this module. Other revisions or emulation schemes may vary in their implementation details and may have different wear-leveling, latency, and other characteristics.

29.2.1.1. Emulator Characteristics

This emulator is designed for **best reliability, with a good balance of available storage and write-cycle limits**. It is designed to ensure that page data is automatically updated so that in the event of a failed update the previous data is not lost (when used correctly). With the exception of a system reset with data cached to the internal write-cache buffer, at most only the latest write to physical non-volatile memory will be lost in the event of a failed write.

This emulator scheme is tuned to give best write-cycle longevity when writes are confined to the same logical EEPROM page (where possible) and when writes across multiple logical EEPROM pages are made in a linear fashion through the entire emulated EEPROM space.

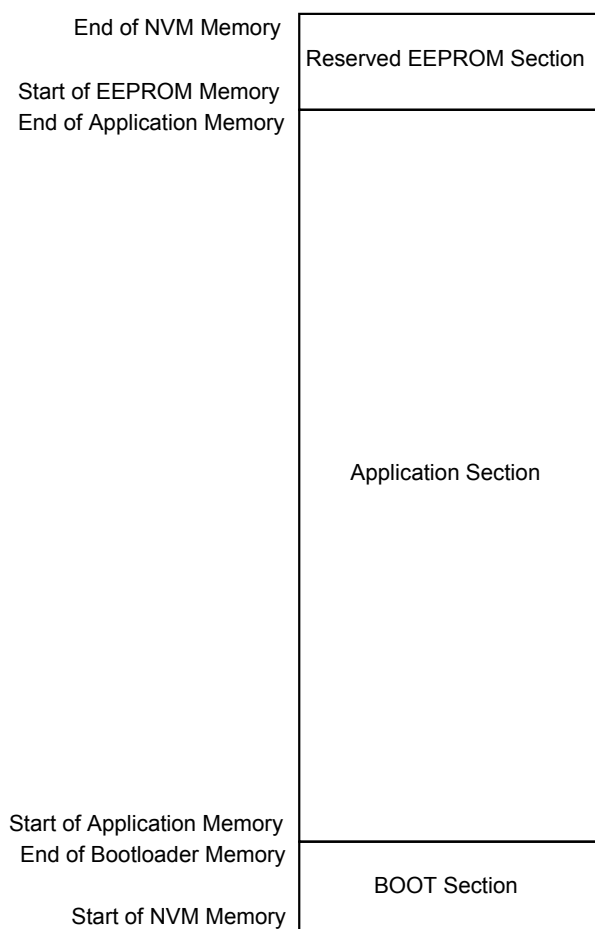
29.2.1.2. Physical Memory

The SAM non-volatile FLASH is divided into a number of physical rows, each containing four identically sized flash pages. Pages may be read or written to individually, however pages must be erased before being reprogrammed and the smallest granularity available for erasure is one single row.

This discrepancy results in the need for an emulator scheme that is able to handle the versioning and moving of page data to different physical rows as needed, erasing old rows ready for re-use by future page write operations.

Physically, the emulated EEPROM segment is located at the end of the physical FLASH memory space, as shown in [Figure 29-1](#).

Figure 29-1. Physical Memory



29.2.1.3. Master Row

One physical FLASH row at the end of the emulated EEPROM memory space is reserved for use by the emulator to store configuration data. The master row is not user-accessible, and is reserved solely for internal use by the emulator.

29.2.1.4. Spare Row

As data needs to be preserved between row erasures, a single FLASH row is kept unused to act as destination for copied data when a write request is made to an already full row. When the write request is made, any logical pages of data in the full row that need to be preserved are written to the spare row along with the new (updated) logical page data, before the old row is erased and marked as the new spare.

29.2.1.5. Row Contents

Each physical FLASH row initially stores the contents of two logical EEPROM memory pages. This halves the available storage space for the emulated EEPROM but reduces the overall number of row erases that are required, by reserving two pages within each row for updated versions of the logical page contents. See [Figure 29-3](#) for a visual layout of the EEPROM Emulator physical memory.

As logical pages within a physical row are updated, the new data is filled into the remaining unused pages in the row. Once the entire row is full, a new write request will copy the logical page not being written to in the current row to the spare row with the new (updated) logical page data, before the old row is erased.

This system allows for the same logical page to be updated up to three times into physical memory before a row erasure procedure is needed. In the case of multiple versions of the same logical EEPROM page being stored in the same physical row, the right-most (highest physical FLASH memory page address) version is considered to be the most current.

29.2.1.6. Write Cache

As a typical EEPROM use case is to write to multiple sections of the same EEPROM page sequentially, the emulator is optimized with a single logical EEPROM page write cache to buffer writes before they are written to the physical backing memory store. The cache is automatically committed when a new write request to a different logical EEPROM memory page is requested, or when the user manually commits the write cache.

Without the write cache, each write request to an EEPROM memory page would require a full page write, reducing the system performance and significantly reducing the lifespan of the non-volatile memory.

29.2.2. Memory Layout

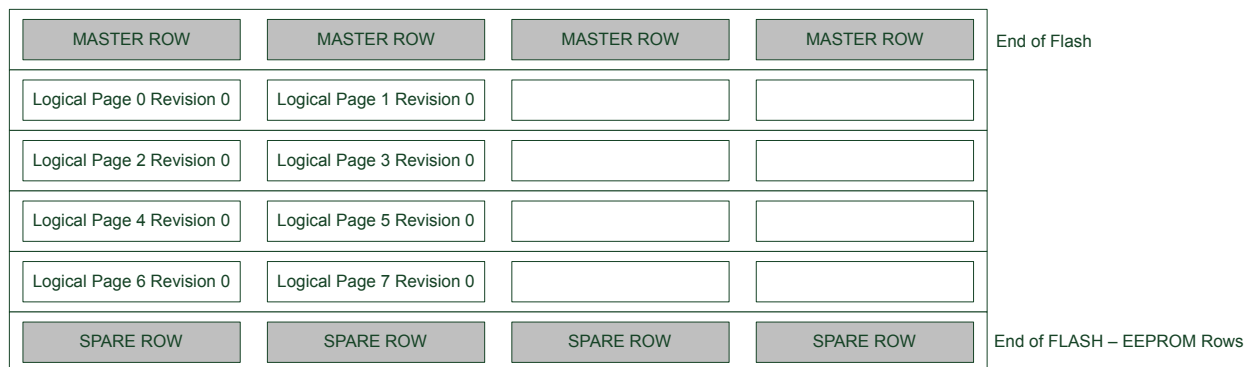
A single logical EEPROM page is physically stored as the page contents and a header inside a single physical FLASH page, as shown in [Figure 29-2](#).

Figure 29-2. Internal Layout of An Emulated EEPROM Page



Within the EEPROM memory reservation section at the top of the NVM memory space, this emulator will produce the layout as shown in [Figure 29-3](#) when initialized for the first time.

Figure 29-3. Initial Physical Layout of The Emulated EEPROM Memory



When an EEPROM page needs to be committed to physical memory, the next free FLASH page in the same row will be chosen - this makes recovery simple, as the right-most version of a logical page in a row is considered the most current. With four pages to a physical NVM row, this allows for up to three updates to the same logical page to be made before an erase is needed. [Figure 29-4](#) shows the result of the user writing an updated version of logical EEPROM page $N-1$ to the physical memory.

Figure 29-4. First Write to Logical EEPROM Page N-1

MASTER ROW	MASTER ROW	MASTER ROW	MASTER ROW	End of Flash
Logical Page 0 Revision 0	Logical Page 1 Revision 0	Logical Page 0 Revision 1		
Logical Page 2 Revision 0	Logical Page 3 Revision 0			
Logical Page 4 Revision 0	Logical Page 5 Revision 0			
Logical Page 6 Revision 0	Logical Page 7 Revision 0			
SPARE ROW	SPARE ROW	SPARE ROW	SPARE ROW	
				End of FLASH – EEPROM Rows

A second write of the same logical EEPROM page results in the layout shown in [Figure 29-5](#).

Figure 29-5. Second Write to Logical EEPROM Page N-1

MASTER ROW	MASTER ROW	MASTER ROW	MASTER ROW	End of Flash
Logical Page 0 Revision 0	Logical Page 1 Revision 0	Logical Page 0 Revision 1	Logical Page 0 Revision 2	
Logical Page 2 Revision 0	Logical Page 3 Revision 0			
Logical Page 4 Revision 0	Logical Page 5 Revision 0			
Logical Page 6 Revision 0	Logical Page 7 Revision 0			
SPARE ROW	SPARE ROW	SPARE ROW	SPARE ROW	
				End of FLASH – EEPROM Rows

A third write of the same logical page requires that the EEPROM emulator erase the row, as it has become full. Prior to this, the contents of the unmodified page in the same row as the page being updated will be copied into the spare row, along with the new version of the page being updated. The old (full) row is then erased, resulting in the layout shown in [Figure 29-6](#).

Figure 29-6. Third Write to Logical EEPROM Page N-1

MASTER ROW	MASTER ROW	MASTER ROW	MASTER ROW	End of Flash
SPARE ROW	SPARE ROW	SPARE ROW	SPARE ROW	
Logical Page 2 Revision 0	Logical Page 3 Revision 0			
Logical Page 4 Revision 0	Logical Page 5 Revision 0			
Logical Page 6 Revision 0	Logical Page 7 Revision 0			
Logical Page 0 Revision 3	Logical Page 1 Revision 0			
				End of FLASH – EEPROM Rows

29.3. Special Considerations

29.3.1. NVM Controller Configuration

The EEPROM Emulator service will initialize the NVM controller as part of its own initialization routine; the NVM controller will be placed in Manual Write mode, so that explicit write commands must be sent to the controller to commit a buffered page to physical memory. The manual write command must thus be

issued to the NVM controller whenever the user application wishes to write to a NVM page for its own purposes.

29.3.2. Logical EEPROM Page Size

As a small amount of information needs to be stored in a header before the contents of a logical EEPROM page in memory (for use by the emulation service), the available data in each EEPROM page is less than the total size of a single NVM memory page by several bytes.

29.3.3. Committing of the Write Cache

A single-page write cache is used internally to buffer data written to pages in order to reduce the number of physical writes required to store the user data, and to preserve the physical memory lifespan. As a result, it is important that the write cache is committed to physical memory **as soon as possible after a BOD low power condition**, to ensure that enough power is available to guarantee a completed write so that no data is lost.

The write cache must also be manually committed to physical memory if the user application is to perform any NVM operations using the NVM controller directly.

29.4. Extra Information

For extra information, see [Extra Information](#). This includes:

- [Acronyms](#)
- [Dependencies](#)
- [Errata](#)
- [Module History](#)

29.5. Examples

For a list of examples related to this driver, see [Examples for Emulated EEPROM Service](#).

29.6. API Overview

29.6.1. Structure Definitions

29.6.1.1. Struct `eeeprom_emulator_parameters`

Structure containing the memory layout parameters of the EEPROM emulator module.

Table 29-1. Members

Type	Name	Description
uint16_t	eeeprom_number_of_pages	Number of emulated pages of EEPROM.
uint8_t	page_size	Number of bytes per emulated EEPROM page.

29.6.2. Macro Definitions

29.6.2.1. EEPROM Emulator Information

Macro EEPROM_EMULATOR_ID

```
#define EEPROM_EMULATOR_ID
```

Emulator scheme ID, identifying the scheme used to emulate EEPROM storage.

Macro EEPROM_MAJOR_VERSION

```
#define EEPROM_MAJOR_VERSION
```

Emulator major version number, identifying the emulator major version.

Macro EEPROM_MINOR_VERSION

```
#define EEPROM_MINOR_VERSION
```

Emulator minor version number, identifying the emulator minor version.

Macro EEPROM_REVISION

```
#define EEPROM_REVISION
```

Emulator revision version number, identifying the emulator revision.

Macro EEPROM_PAGE_SIZE

```
#define EEPROM_PAGE_SIZE
```

Size of the user data portion of each logical EEPROM page, in bytes.

29.6.3. Function Definitions

29.6.3.1. Configuration and Initialization

Function eeprom_emulator_init()

Initializes the EEPROM Emulator service.

```
enum status_code eeprom_emulator_init( void )
```

Initializes the emulated EEPROM memory space; if the emulated EEPROM memory has not been previously initialized, it will need to be explicitly formatted via [eeprom_emulator_erase_memory\(\)](#). The EEPROM memory space will **not** be automatically erased by the initialization function, so that partial data may be recovered by the user application manually if the service is unable to initialize successfully.

Returns

Status code indicating the status of the operation.

Table 29-2. Return Values

Return value	Description
STATUS_OK	EEPROM emulation service was successfully initialized
STATUS_ERR_NO_MEMORY	No EEPROM section has been allocated in the device

Return value	Description
STATUS_ERR_BAD_FORMAT	Emulated EEPROM memory is corrupt or not formatted
STATUS_ERR_IO	EEPROM data is incompatible with this version or scheme of the EEPROM emulator

Function `eeeprom_emulator_erase_memory()`

Erases the entire emulated EEPROM memory space.

```
void eeeprom_emulator_erase_memory( void )
```

Erases and re-initializes the emulated EEPROM memory space, destroying any existing data.

Function `eeeprom_emulator_get_parameters()`

Retrieves the parameters of the EEPROM Emulator memory layout.

```
enum status_code eeeprom_emulator_get_parameters(
    struct eeeprom_emulator_parameters *const parameters)
```

Retrieves the configuration parameters of the EEPROM Emulator, after it has been initialized.

Table 29-3. Parameters

Data direction	Parameter name	Description
[out]	parameters	EEPROM Emulator parameter struct to fill

Returns

Status of the operation.

Table 29-4. Return Values

Return value	Description
STATUS_OK	If the emulator parameters were retrieved successfully
STATUS_ERR_NOT_INITIALIZED	If the EEPROM Emulator is not initialized

29.6.3.2. Logical EEPROM Page Reading/Writing

Function `eeeprom_emulator_commit_page_buffer()`

Commits any cached data to physical non-volatile memory.

```
enum status_code eeeprom_emulator_commit_page_buffer( void )
```

Commits the internal SRAM caches to physical non-volatile memory, to ensure that any outstanding cached data is preserved. This function should be called prior to a system reset or shutdown to prevent data loss.

Note: This should be the first function executed in a BOD33 Early Warning callback to ensure that any outstanding cache data is fully written to prevent data loss.

Note: This function should also be called before using the NVM controller directly in the user-application for any other purposes to prevent data loss.

Returns

Status code indicating the status of the operation.

Function `eeeprom_emulator_write_page()`

Writes a page of data to an emulated EEPROM memory page.

```
enum status_code eeeprom_emulator_write_page(  
    const uint8_t logical_page,  
    const uint8_t *const data)
```

Writes an emulated EEPROM page of data to the emulated EEPROM memory space.

Note: Data stored in pages may be cached in volatile RAM memory; to commit any cached data to physical non-volatile memory, the `eeeprom_emulator_commit_page_buffer()` function should be called.

Table 29-5. Parameters

Data direction	Parameter name	Description
[in]	logical_page	Logical EEPROM page number to write to
[in]	data	Pointer to the data buffer containing source data to write

Returns

Status code indicating the status of the operation.

Table 29-6. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

Function `eeeprom_emulator_read_page()`

Reads a page of data from an emulated EEPROM memory page.

```
enum status_code eeeprom_emulator_read_page(  
    const uint8_t logical_page,  
    uint8_t *const data)
```

Reads an emulated EEPROM page of data from the emulated EEPROM memory space.

Table 29-7. Parameters

Data direction	Parameter name	Description
[in]	logical_page	Logical EEPROM page number to read from
[out]	data	Pointer to the destination data buffer to fill

Returns

Status code indicating the status of the operation.

Table 29-8. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

29.6.3.3. Buffer EEPROM Reading/Writing

Function `eeeprom_emulator_write_buffer()`

Writes a buffer of data to the emulated EEPROM memory space.

```
enum status_code eeeprom_emulator_write_buffer(
    const uint16_t offset,
    const uint8_t *const data,
    const uint16_t length)
```

Writes a buffer of data to a section of emulated EEPROM memory space. The source buffer may be of any size, and the destination may lie outside of an emulated EEPROM page boundary.

Note: Data stored in pages may be cached in volatile RAM memory; to commit any cached data to physical non-volatile memory, the `eeeprom_emulator_commit_page_buffer()` function should be called.

Table 29-9. Parameters

Data direction	Parameter name	Description
[in]	offset	Starting byte offset to write to, in emulated EEPROM memory space
[in]	data	Pointer to the data buffer containing source data to write
[in]	length	Length of the data to write, in bytes

Returns

Status code indicating the status of the operation.

Table 29-10. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

Function `eeeprom_emulator_read_buffer()`

Reads a buffer of data from the emulated EEPROM memory space.

```
enum status_code eeeprom_emulator_read_buffer(
    const uint16_t offset,
    uint8_t *const data,
    const uint16_t length)
```

Reads a buffer of data from a section of emulated EEPROM memory space. The destination buffer may be of any size, and the source may lie outside of an emulated EEPROM page boundary.

Table 29-11. Parameters

Data direction	Parameter name	Description
[in]	offset	Starting byte offset to read from, in emulated EEPROM memory space
[out]	data	Pointer to the data buffer containing source data to read
[in]	length	Length of the data to read, in bytes

Returns

Status code indicating the status of the operation.

Table 29-12. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated EEPROM memory space was supplied

29.7. Extra Information

29.7.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
EEPROM	Electrically Erasable Read-Only Memory
NVM	Non-Volatile Memory

29.7.2. Dependencies

This driver has the following dependencies:

- Non-Volatile Memory Controller Driver

29.7.3. Errata

There are no errata related to this driver.

29.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog

Fix warnings

Initial Release

29.8. Examples for Emulated EEPROM Service

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM EEPROM Emulator \(EEPROM\) Service](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that QSGs can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for the Emulated EEPROM Module - Basic Use Case](#)

29.8.1. Quick Start Guide for the Emulated EEPROM Module - Basic Use Case

In this use case, the EEPROM emulator module is configured and a sample page of data read and written. The first byte of the first EEPROM page is toggled, and a LED is turned on or off to reflect the new state. Each time the device is reset, the LED should toggle to a different state to indicate correct non-volatile storage and retrieval.

29.8.1.1. Prerequisites

The device's fuses must be configured to reserve a sufficient number of FLASH memory rows for use by the EEPROM emulator service, before the service can be used. That is:

NVMCTRL_FUSES_EEPROM_SIZE has to be set to less than 0x5 in the fuse setting, then there will be more than 8 pages size for EEPROM. Atmel Studio can be used to set this fuse (Tools->Device Programming).

29.8.1.2. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_eeprom(void)
{
    /* Setup EEPROM emulator service */
    enum status_code error_code = eeprom_emulator_init();

    if (error_code == STATUS_ERR_NO_MEMORY) {
        while (true) {
            /* No EEPROM section has been set in the device's fuses */
        }
    }
    else if (error_code != STATUS_OK) {
        /* Erase the emulated EEPROM memory (assume it is unformatted or
        * irrecoverably corrupt) */
        eeprom_emulator_erase_memory();
        eeprom_emulator_init();
    }
}

#if (SAMD || SAMR21)
```

```

void SYSCTRL_Handler(void)
{
    if (SYSCTRL->INTFLAG.reg & SYSCTRL_INTFLAG_BOD33DET) {
        SYSCTRL->INTFLAG.reg = SYSCTRL_INTFLAG_BOD33DET;
        eeprom_emulator_commit_page_buffer();
    }
}
#endif
static void configure_bod(void)
{
    #if (SAM_D || SAM_R21)
        struct bod_config config_bod33;
        bod_get_config_defaults(&config_bod33);
        config_bod33.action = BOD_ACTION_INTERRUPT;
        /* BOD33 threshold level is about 3.2V */
        config_bod33.level = 48;
        bod_set_config(BOD_BOD33, &config_bod33);
        bod_enable(BOD_BOD33);

        SYSCTRL->INTENSET.reg = SYSCTRL_INTENCLR_BOD33DET;
        system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_SYSCTRL);
    #endif
}

```

Add to user application initialization (typically the start of `main()`):

```
configure_eeprom();
```

Workflow

1. Attempt to initialize the EEPROM emulator service, storing the error code from the initialization function into a temporary variable.
2. Check if the emulator failed to initialize due to the device fuses not being configured to reserve enough of the main FLASH memory rows for emulated EEPROM usage - abort if the fuses are mis-configured.

```

enum status_code error_code = eeprom_emulator_init();

if (error_code == STATUS_ERR_NO_MEMORY) {
    while (true) {
        /* No EEPROM section has been set in the device's fuses */
    }
}

```

3. Check if the emulator service failed to initialize for any other reason; if so assume the emulator physical memory is unformatted or corrupt and erase/re-try initialization.

```

else if (error_code != STATUS_OK) {
    /* Erase the emulated EEPROM memory (assume it is unformatted or
    * irrecoverably corrupt) */
    eeprom_emulator_erase_memory();
    eeprom_emulator_init();
}

```

Config BOD to give an early warning, so that we could prevent data loss.

```
configure_bod();
```

29.8.1.3. Use Case

Code

Copy-paste the following code to your user application:

```
uint8_t page_data[EEPROM_PAGE_SIZE];
eeprom_emulator_read_page(0, page_data);

page_data[0] = !page_data[0];
port_pin_set_output_level(LED_0_PIN, page_data[0]);

eeprom_emulator_write_page(0, page_data);
eeprom_emulator_commit_page_buffer();

page_data[1]=0x1;
eeprom_emulator_write_page(0, page_data);

while (true) {
}
```

Workflow

1. Create a buffer to hold a single emulated EEPROM page of memory, and read out logical EEPROM page zero into it.

```
uint8_t page_data[EEPROM_PAGE_SIZE];
eeprom_emulator_read_page(0, page_data);
```

2. Toggle the first byte of the read page.

```
page_data[0] = !page_data[0];
```

3. Output the toggled LED state onto the board LED.

```
port_pin_set_output_level(LED_0_PIN, page_data[0]);
```

4. Write the modified page back to logical EEPROM page zero, flushing the internal emulator write cache afterwards to ensure it is immediately written to physical non-volatile memory.

```
eeprom_emulator_write_page(0, page_data);
eeprom_emulator_commit_page_buffer();
```

5. Modify data and write back to logical EEPROM page zero. The data is not committed and should call `eeprom_emulator_commit_page_buffer` to ensure that any outstanding cache data is fully written to prevent data loss when detecting a BOD early warning.

```
page_data[1]=0x1;
eeprom_emulator_write_page(0, page_data);
```

30. SAM Read While Write EEPROM (RWW EEPROM) Emulator Service

This driver for Atmel® | SMART ARM®-based microcontrollers provides an RWW emulated EEPROM memory area, for the storage and retrieval of user-application configuration data into and out of non-volatile memory. The main array can therefore run code while EEPROM data is written.

The following peripheral is used by this module:

- NVM (Non-Volatile Memory Controller)

The following devices can use this module:

- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM D21
- Atmel | SMART SAM C20/C21
- Atmel | SMART SAM DA1

The outline of this documentation is as follows:

- [Prerequisites](#)
- [Module Overview](#)
- [Special Considerations](#)
- [Extra Information](#)
- [Examples](#)
- [API Overview](#)

30.1. Prerequisites

There are no prerequisites for this module.

30.2. Module Overview

SAM devices embeds a separate read while write EEPROM emulation (RWWEE) array that can be programmed while the main array is not blocked. To use RWWEE memory, data must be written as a number of physical memory pages (of several bytes each) rather than being individually byte addressable, and entire rows of RWWEE must be erased before new data may be stored. To help abstract these characteristics away from the user application an emulation scheme is implemented to present a more user-friendly API for data storage and retrieval.

This module provides an RWW EEPROM emulation layer on top of the device's internal NVM controller, to provide a standard interface for the reading and writing of non-volatile configuration data. This data is placed into the RWW EEPROM emulated section. Emulated EEPROM is exempt from the usual device NVM region lock bits, so that it may be read from or written to at any point in the user application.

There are many different algorithms that may be employed for EEPROM emulation, to tune the write and read latencies, RAM usage, wear levelling, and other characteristics. As a result, multiple different emulator schemes may be implemented, so that the most appropriate scheme for a specific application's requirements may be used.

30.2.1. Implementation Details

The following information is relevant for **RWW EEPROM Emulator scheme 1, version 1.0.0**, as implemented by this module. Other revisions or emulation schemes may vary in their implementation details and may have different wear-leveling, latency, and other characteristics.

30.2.1.1. Emulator Characteristics

This emulator is designed for **best reliability, with a good balance of available storage and write-cycle limits**. It is designed to ensure that page data is updated by an atomic operation, so that in the event of a failed update the previous data is not lost (when used correctly). With the exception of a system reset with data cached to the internal write-cache buffer, at most only the latest write to physical non-volatile memory will be lost in the event of a failed write.

This emulator scheme is tuned to give best write-cycle longevity when writes are confined to the same logical RWW EEPROM page (where possible) and when writes across multiple logical RWW EEPROM pages are made in a linear fashion through the entire emulated RWW EEPROM space.

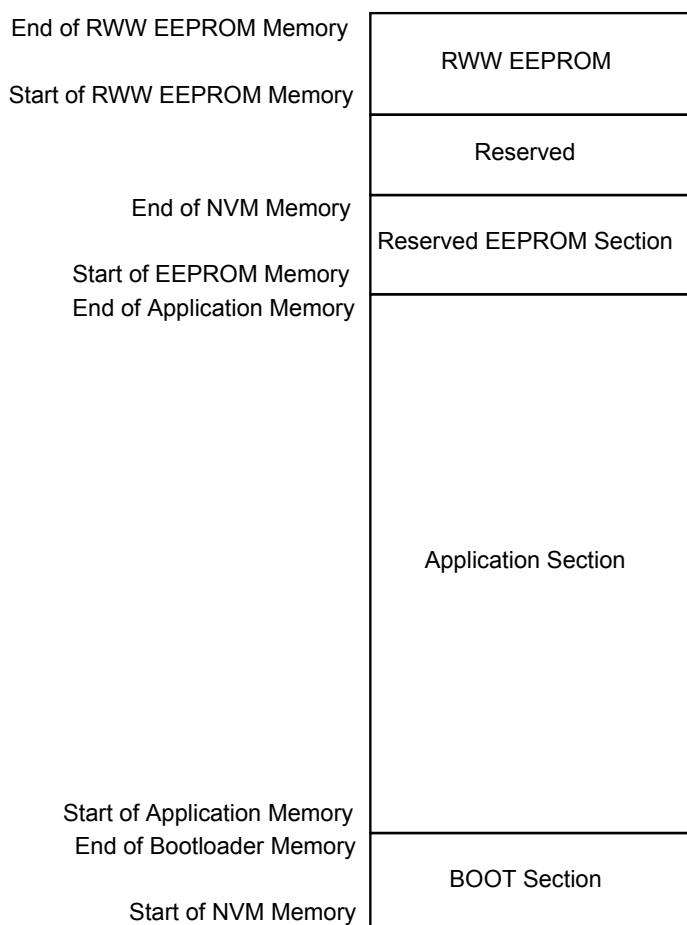
30.2.1.2. Physical Memory

RWW EEPROM emulator is divided into a number of physical rows, each containing four identically sized pages. Pages may be read or written to individually, however, pages must be erased before being reprogrammed and the smallest granularity available for erasure is one single row.

This discrepancy results in the need for an emulator scheme that is able to handle the versioning and moving of page data to different physical rows as needed, erasing old rows ready for re-use by future page write operations.

Physically, the emulated RWW EEPROM segment is a dedicated space that are memory mapped, as shown in [Figure 30-1](#).

Figure 30-1. Physical Memory



30.2.1.3. Master Row

One physical row at the end of the emulated RWW EEPROM memory space is reserved for use by the emulator to store configuration data. The master row is not user-accessible, and is reserved solely for internal use by the emulator.

30.2.1.4. Spare Row

As data needs to be preserved between row erasures, a single row is kept unused to act as destination for copied data when a write request is made to an already full row. When the write request is made, any logical pages of data in the full row that need to be preserved are written to the spare row along with the new (updated) logical page data, before the old row is erased and marked as the new spare.

30.2.1.5. Row Contents

Each physical row initially stores the contents of one or two logical RWW EEPROM memory pages (it depends on application configuration file). This quarters or halves the available storage space for the emulated RWW EEPROM but reduces the overall number of row erases that are required, by reserving two or three pages within each row for updated versions of the logical page contents. See [Figure 30-2](#) for a visual layout of the RWW EEPROM Emulator physical memory.

As logical pages within a physical row are updated, the new data is filled into the remaining unused pages in the row. Once the entire row is full, a new write request will copy the logical page not being written to in the current row to the spare row with the new (updated) logical page data, before the old row is erased.

When it is configured, each physical row stores the contents of one logical RWW EEPROM memory page. This system will allow for the same logical page to be updated up to four times into the physical memory before a row erasure procedure is needed. In the case of multiple versions of the same logical RWW EEPROM page being stored in the same physical row, the right-most (highest physical memory page address) version is considered to be the most current.

30.2.1.6. Write Cache

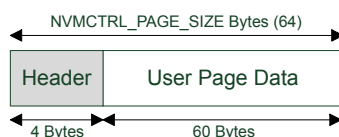
As a typical EEPROM use case is to write to multiple sections of the same EEPROM page sequentially, the emulator is optimized with a single logical RWW EEPROM page write cache to buffer writes before they are written to the physical backing memory store. The cache is automatically committed when a new write request to a different logical RWW EEPROM memory page is requested, or when the user manually commits the write cache.

Without the write cache, each write request to an EEPROM memory page would require a full page write, reducing the system performance and significantly reducing the lifespan of the non-volatile memory.

30.2.2. Memory Layout

A single logical RWW EEPROM page is physically stored as the page content and a header inside a single physical page, as shown in [Figure 30-2](#).

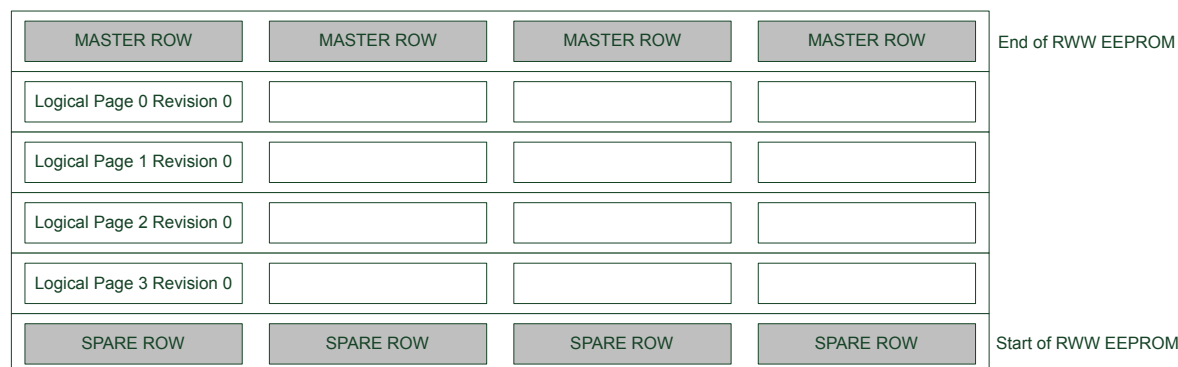
Figure 30-2. Internal Layout of an Emulated RWW EEPROM Page



Note: In the following memory layout example, each physical row stores the contents of one logical RWW EEPROM page. Refer to ["AT03265: SAM EEPROM Emulator Service \(EEPROM\)"](#) for the example of two logical EEPROM pages in one row.

Within the RWW EEPROM memory reservation section at the top of the NVM memory space, this emulator will produce the layout as shown in [Figure 30-3](#) when initialized for the first time.

Figure 30-3. Initial Physical Layout of the Emulated RWW EEPROM Memory



When an RWW EEPROM page needs to be committed to physical memory, the next free page in the same row will be chosen. This makes recovery simple, as the right-most version of a logical page in a row is considered the most current. With four pages to a physical NVM row, this allows for up to four updates to the same logical page to be made before an erase is needed. [Figure 30-4](#) shows the result of the user writing an updated version of logical EEPROM page $N-1$ to the physical memory.

Figure 30-4. First Write to Logical RWW EEPROM Page N-1

MASTER ROW	MASTER ROW	MASTER ROW	MASTER ROW	End of RWW EEPROM
Logical Page 0 Revision 0	Logical Page 0 Revision 1			
Logical Page 1 Revision 0				
Logical Page 2 Revision 0				
Logical Page 3 Revision 0				
SPARE ROW	SPARE ROW	SPARE ROW	SPARE ROW	Start of RWW EEPROM

A second write of the same logical RWW EEPROM page results in the layout shown in [Figure 30-5](#).

Figure 30-5. Second Write to Logical RWW EEPROM Page N-1

MASTER ROW	MASTER ROW	MASTER ROW	MASTER ROW	End of RWW EEPROM
Logical Page 0 Revision 0	Logical Page 0 Revision 1	Logical Page 0 Revision 2		
Logical Page 1 Revision 0				
Logical Page 2 Revision 0				
Logical Page 3 Revision 0				
SPARE ROW	SPARE ROW	SPARE ROW	SPARE ROW	Start of RWW EEPROM

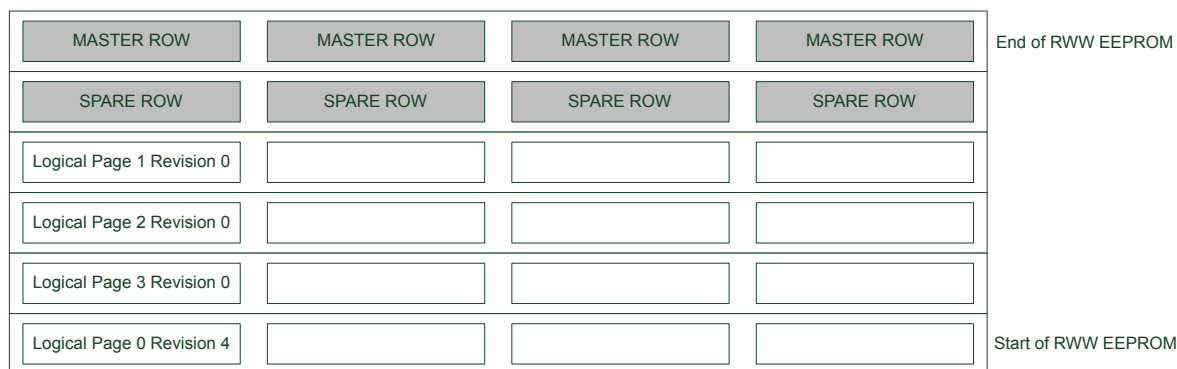
A third write of the same logical RWW EEPROM page results in the layout shown in [Figure 30-6](#).

Figure 30-6. Third Write to Logical RWW EEPROM Page N-1

MASTER ROW	MASTER ROW	MASTER ROW	MASTER ROW	End of RWW EEPROM
Logical Page 0 Revision 0	Logical Page 0 Revision 1	Logical Page 0 Revision 2	Logical Page 0 Revision 3	
Logical Page 1 Revision 0				
Logical Page 2 Revision 0				
Logical Page 3 Revision 0				
SPARE ROW	SPARE ROW	SPARE ROW	SPARE ROW	Start of RWW EEPROM

A fourth write of the same logical page requires that the RWW EEPROM emulator erase the row, as it has become full. Prior to this, the content of the unmodified page in the same row as the page being updated will be copied into the spare row, along with the new version of the page being updated. The old (full) row is then erased, resulting in the layout shown in [Figure 30-7](#).

Figure 30-7. Third Write to Logical RWW EEPROM Page N-1



30.3. Special Considerations

30.3.1. NVM Controller Configuration

The RWW EEPROM Emulator service will initialize the NVM controller as part of its own initialization routine; the NVM controller will be placed in Manual Write mode, so that explicit write commands must be sent to the controller to commit a buffered page to physical memory. The manual write command must thus be issued to the NVM controller whenever the user application wishes to write to a NVM page for its own purposes.

30.3.2. Logical RWW EEPROM Page Size

As a small amount of information needs to be stored in a header before the content of a logical EEPROM page in memory (for use by the emulation service), the available data in each RWW EEPROM page is less than the total size of a single NVM memory page by several bytes.

30.3.3. Committing of the Write Cache

A single-page write cache is used internally to buffer data written to pages in order to reduce the number of physical writes required to store the user data, and to preserve the physical memory lifespan. As a result, it is important that the write cache is committed to physical memory **as soon as possible after a BOD low power condition**, to ensure that enough power is available to guarantee a completed write so that no data is lost.

The write cache must also be manually committed to physical memory if the user application is to perform any NVM operations using the NVM controller directly.

30.3.4. RWW EEPROM Page Checksum

For each page, a checksum function is used to verify the integrity of the page data. When reading the page data, using [rww_eeprom_emulator_read_page\(\)](#). When its checksum is not correct, an error can be detected. This function can be enabled or disabled through the configuration file.

30.4. Extra Information

For extra information, see [Extra Information](#). This includes:

- [Acronyms](#)
- [Dependencies](#)

- [Errata](#)
- [Module History](#)

30.5. Examples

For a list of examples related to this driver, see [Examples for Emulated RWW EEPROM Service](#).

30.6. API Overview

30.6.1. Structure Definitions

30.6.1.1. Struct `rww_eeprom_emulator_parameters`

Structure containing the memory layout parameters of the EEPROM emulator module.

Table 30-1. Members

Type	Name	Description
uint16_t	eeprom_number_of_pages	Number of emulated pages of EEPROM
uint8_t	page_size	Number of bytes per emulated EEPROM page

30.6.2. Macro Definitions

30.6.2.1. RWW EEPROM Emulator Information

Macro `RWW_EEPROM_EMULATOR_ID`

```
#define RWW_EEPROM_EMULATOR_ID
```

Emulator scheme ID, identifying the scheme used to emulated EEPROM storage.

Macro `RWW_EEPROM_MAJOR_VERSION`

```
#define RWW_EEPROM_MAJOR_VERSION
```

Emulator major version number, identifying the emulator major version.

Macro `RWW_EEPROM_MINOR_VERSION`

```
#define RWW_EEPROM_MINOR_VERSION
```

Emulator minor version number, identifying the emulator minor version.

Macro `RWW_EEPROM_REVISION`

```
#define RWW_EEPROM_REVISION
```

Emulator revision version number, identifying the emulator revision.

Macro `RWW_EEPROM_PAGE_SIZE`

```
#define RWW_EEPROM_PAGE_SIZE
```

Size of the user data portion of each logical EEPROM page, in bytes.

30.6.3. Function Definitions

30.6.3.1. Configuration and Initialization

Function `rw_w_ee_prom_emulator_init()`

Initializes the RWW EEPROM Emulator service.

```
enum status_code rw_w_ee_prom_emulator_init( void )
```

Initializes the emulated RWW EEPROM memory space. If the emulated RWW EEPROM memory has not been previously initialized, it will need to be explicitly formatted via `rw_w_ee_prom_emulator_erase_memory()`. The RWW EEPROM memory space will **not** be automatically erased by the initialization function. Partial data may be recovered by the user application manually if the service is unable to initialize successfully.

Returns

Status code indicating the status of the operation.

Table 30-2. Return Values

Return value	Description
STATUS_OK	RWW EEPROM emulation service was successfully initialized
STATUS_ERR_BAD_FORMAT	Emulated RWW EEPROM memory is corrupt or not formatted
STATUS_ERR_IO	RWW EEPROM data is incompatible with this version or scheme of the RWW EEPROM emulator
STATUS_ERR_INVALID_ARG	Invalid logical page configuration

Function `rw_w_ee_prom_emulator_erase_memory()`

Erases the entire emulated RWW EEPROM memory space.

```
void rw_w_ee_prom_emulator_erase_memory( void )
```

Erases and re-initializes the emulated RWW EEPROM memory space, destroying any existing data.

Function `rw_w_ee_prom_emulator_get_parameters()`

Retrieves the parameters of the RWW EEPROM Emulator memory layout.

```
enum status_code rw_w_ee_prom_emulator_get_parameters(  
    struct rw_w_ee_prom_emulator_parameters *const parameters)
```

Retrieves the configuration parameters of the RWW EEPROM Emulator, after it has been initialized.

Table 30-3. Parameters

Data direction	Parameter name	Description
[out]	parameters	RWW EEPROM Emulator parameter struct to fill

Returns

Status of the operation.

Table 30-4. Return Values

Return value	Description
STATUS_OK	If the emulator parameters were retrieved successfully
STATUS_ERR_NOT_INITIALIZED	If the RWW EEPROM Emulator is not initialized

30.6.3.2. Logical RWW EEPROM Page Reading/Writing

Function `rw_w_eeeprom_emulator_commit_page_buffer()`

Commits any cached data to physical non-volatile memory.

```
enum status_code rw_w_eeeprom_emulator_commit_page_buffer( void )
```

Commits the internal SRAM caches to physical non-volatile memory, to ensure that any outstanding cached data is preserved. This function should be called prior to a system reset or shutdown to prevent data loss.

Note: This should be the first function executed in a BOD33 Early Warning callback to ensure that any outstanding cache data is fully written to prevent data loss.

Note: This function should also be called before using the NVM controller directly in the user-application for any other purposes to prevent data loss.

Returns

Status code indicating the status of the operation.

Function `rw_w_eeeprom_emulator_write_page()`

Writes a page of data to an emulated RWW EEPROM memory page.

```
enum status_code rw_w_eeeprom_emulator_write_page(
    const uint8_t logical_page,
    const uint8_t *const data)
```

Writes an emulated RWW EEPROM page of data to the emulated RWW EEPROM memory space.

Note: Data stored in pages may be cached in volatile RAM memory; to commit any cached data to physical non-volatile memory, the `rw_w_eeeprom_emulator_commit_page_buffer()` function should be called.

Table 30-5. Parameters

Data direction	Parameter name	Description
[in]	logical_page	Logical RWW EEPROM page number to write to
[in]	data	Pointer to the data buffer containing source data to write

Returns

Status code indicating the status of the operation.

Table 30-6. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the RWW EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated RWW EEPROM memory space was supplied

Function `rw_w_ee_prom_emulator_read_page()`

Reads a page of data from an emulated RWW EEPROM memory page.

```
enum status_code rw_w_ee_prom_emulator_read_page(
    const uint8_t logical_page,
    uint8_t *const data)
```

Reads an emulated RWW EEPROM page of data from the emulated RWW EEPROM memory space.

Table 30-7. Parameters

Data direction	Parameter name	Description
[in]	logical_page	Logical RWW EEPROM page number to read from
[out]	data	Pointer to the destination data buffer to fill

Returns

Status code indicating the status of the operation.

Table 30-8. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the RWW EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated RWW EEPROM memory space was supplied
STATUS_ERR_BAD_FORMAT	Page data checksum is not correct, maybe data is damaged

30.6.3.3. Buffer RWW EEPROM Reading/Writing**Function `rw_w_ee_prom_emulator_write_buffer()`**

Writes a buffer of data to the emulated RWW EEPROM memory space.

```
enum status_code rw_w_ee_prom_emulator_write_buffer(
    const uint16_t offset,
    const uint8_t *const data,
    const uint16_t length)
```

Writes a buffer of data to a section of emulated RWW EEPROM memory space. The source buffer may be of any size, and the destination may lie outside of an emulated RWW EEPROM page boundary.

Note: Data stored in pages may be cached in volatile RAM memory; to commit any cached data to physical non-volatile memory, the `rw_w_eeeprom_emulator_commit_page_buffer()` function should be called.

Table 30-9. Parameters

Data direction	Parameter name	Description
[in]	offset	Starting byte offset to write to, in emulated RWW EEPROM memory space
[in]	data	Pointer to the data buffer containing source data to write
[in]	length	Length of the data to write, in bytes

Returns

Status code indicating the status of the operation.

Table 30-10. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the RWW EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated RWW EEPROM memory space was supplied

Function `rw_w_eeeprom_emulator_read_buffer()`

Reads a buffer of data from the emulated RWW EEPROM memory space.

```
enum status_code rw_w_eeeprom_emulator_read_buffer(
    const uint16_t offset,
    uint8_t *const data,
    const uint16_t length)
```

Reads a buffer of data from a section of emulated RWW EEPROM memory space. The destination buffer may be of any size, and the source may lie outside of an emulated RWW EEPROM page boundary.

Table 30-11. Parameters

Data direction	Parameter name	Description
[in]	offset	Starting byte offset to read from, in emulated RWW EEPROM memory space
[out]	data	Pointer to the data buffer containing source data to read
[in]	length	Length of the data to read, in bytes

Returns

Status code indicating the status of the operation.

Table 30-12. Return Values

Return value	Description
STATUS_OK	If the page was successfully read
STATUS_ERR_NOT_INITIALIZED	If the RWW EEPROM emulator is not initialized
STATUS_ERR_BAD_ADDRESS	If an address outside the valid emulated RWW EEPROM memory space was supplied

30.6.4. Enumeration Definitions

30.6.4.1. Enum `rwwee_logical_page_num_in_row`

Enum for the possible logical pages that are stored in each physical row.

Table 30-13. Members

Enum value	Description
RWWE_LOGICAL_PAGE_NUM_1	One logical page stored in a physical row
RWWE_LOGICAL_PAGE_NUM_2	Two logical pages stored in a physical row

30.7. Extra Information

30.7.1. Acronyms

Acronym	Description
EEPROM	Electrically Erasable Read-Only Memory
RWWE	Read While Write EEPROM
RWW	Read While Write
NVM	Non-Volatile Memory

30.7.2. Dependencies

This driver has the following dependencies:

- Non-Volatile Memory Controller Driver

30.7.3. Errata

There are no errata related to this driver.

30.7.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog

Initial Release

30.8. Examples for Emulated RWW EEPROM Service

This is a list of the available Quick Start guides (QSGs) and example applications for [SAM Read While Write EEPROM \(RWW EEPROM\) Emulator Service](#). QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- [Quick Start Guide for the Emulated RWW EEPROM Module - Basic Use Case](#)

30.8.1. Quick Start Guide for the Emulated RWW EEPROM Module - Basic Use Case

In this use case, the RWW EEPROM emulator module is configured, and a sample page is read and written. The first byte of the first RWW EEPROM page is toggled, and a LED is turned ON or OFF to reflect the new state. Each time the device is reset, the LED should toggle to a different state to indicate correct non-volatile storage and retrieval.

30.8.1.1. Setup

Prerequisites

There are no special setup requirements for this use-case.

Code

Copy-paste the following setup code to your user application:

```
void configure_eeprom(void)
{
    /* Setup EEPROM emulator service */
    enum status_code error_code = rww_eeprom_emulator_init();

    if (error_code == STATUS_ERR_NO_MEMORY) {
        while (true) {
            /* No EEPROM section has been set in the device's fuses */
        }
    }
    else if (error_code != STATUS_OK) {
        /* Erase the emulated EEPROM memory (assume it is unformatted or
        * irrecoverably corrupt) */
        rww_eeprom_emulator_erase_memory();
        rww_eeprom_emulator_init();
    }
}

#if (SAMD21) || (SAMD1)
void SYSCTRL_Handler(void)
{
    if (SYSCTRL->INTFLAG.reg & SYSCTRL_INTFLAG_BOD33DET) {
        SYSCTRL->INTFLAG.reg |= SYSCTRL_INTFLAG_BOD33DET;
        rww_eeprom_emulator_commit_page_buffer();
    }
}
#endif
static void configure_bod(void)
{
    #if (SAMD21) || (SAMD1)
```



```

struct bod_config config_bod33;
bod_get_config_defaults(&config_bod33);
config_bod33.action = BOD_ACTION_INTERRUPT;
/* BOD33 threshold level is about 3.2V */
config_bod33.level = 48;
bod_set_config(BOD_BOD33, &config_bod33);
bod_enable(BOD_BOD33);

SYSCTRL->INTENSET.reg |= SYSCTRL_INTENCLR_BOD33DET;
system_interrupt_enable(SYSTEM_INTERRUPT_MODULE_SYSCTRL);
#endif
}

```

Add to user application initialization (typically the start of `main()`):

```
configure_eeprom();
```

Workflow

1. Attempt to initialize the RWW EEPROM emulator service, storing the error code from the initialization function into a temporary variable.
2. Check if the emulator service failed to initialize for any other reason; if so, assume the emulator physical memory is unformatted or corrupt and erase/re-try initialization.

```
enum status_code error_code = rww_eeprom_emulator_init();
```

```

else if (error_code != STATUS_OK) {
    /* Erase the emulated EEPROM memory (assume it is unformatted or
    * irrecoverably corrupt) */
    rww_eeprom_emulator_erase_memory();
    rww_eeprom_emulator_init();
}

```

Config BOD to give an early warning to prevent data loss.

```
configure_bod();
```

30.8.1.2. Use Case

Code

Copy-paste the following code to your user application:

```

uint8_t page_data[RWW_EEPROM_PAGE_SIZE];
rww_eeprom_emulator_read_page(0, page_data);

page_data[0] = !page_data[0];
port_pin_set_output_level(LED_0_PIN, page_data[0]);

rww_eeprom_emulator_write_page(0, page_data);
rww_eeprom_emulator_commit_page_buffer();

page_data[1]=0x1;
rww_eeprom_emulator_write_page(0, page_data);

while (true) {
}

```

Workflow

1. Create a buffer to hold a single emulated RWW EEPROM page of memory, and read out logical RWW EEPROM page zero into it.

```
uint8_t page_data[RWW_EEPROM_PAGE_SIZE];  
rww_eeprom_emulator_read_page(0, page_data);
```

2. Toggle the first byte of the read page.

```
page_data[0] = !page_data[0];
```

3. Output the toggled LED state onto the board LED.

```
port_pin_set_output_level(LED_0_PIN, page_data[0]);
```

4. Write the modified page back to logical RWW EEPROM page zero, flushing the internal emulator write cache afterwards to ensure it is immediately written to physical non-volatile memory.

```
rww_eeprom_emulator_write_page(0, page_data);  
rww_eeprom_emulator_commit_page_buffer();
```

5. Modify data and write back to logical EEPROM page zero. The data is not committed and should call `rww_eeprom_emulator_commit_page_buffer` to ensure that any outstanding cache data is fully written to prevent data loss when detecting a BOD early warning.

```
page_data[1]=0x1;  
rww_eeprom_emulator_write_page(0, page_data);
```

31. Document Revision History

Doc. Rev.	Date	Comments
42697A	08/2016	Initial document release.



Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA T: (+1)(408) 441.0311 F: (+1)(408) 436.4200 | www.atmel.com

© 2016 Atmel Corporation. / Rev.: Atmel-42697A-ASF-Manual-SAM-L22_AT13910_Application Note-08/2016

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected®, Cortex®, and others are registered trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death ("Safety-Critical Applications") without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.