APPLICATION NOTE

Atmel AT03665: ASF Manual (SAM D20)

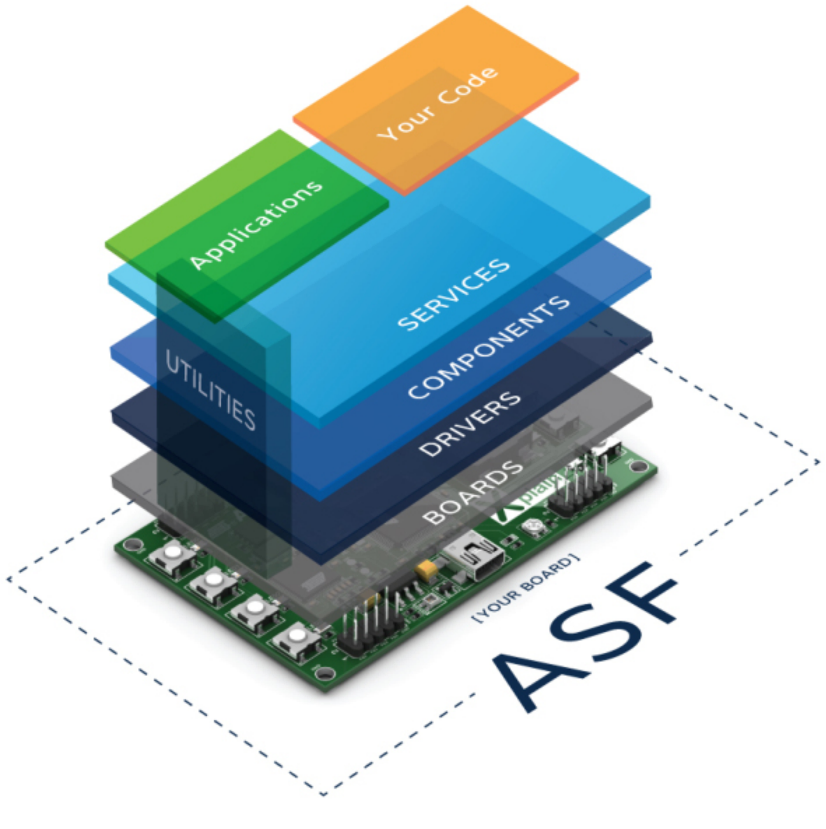
ASF PROGRAMMERS MANUAL

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 please refer to the online documentation at www.atmel.com/asf.

Table of Contents

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 followinFcong 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 D20 Analog Comparator Driver (AC)

This driver for SAM D20 devices 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 outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for AC
- 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 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.

2.2.2 Positive and Negative Input MUXs

Each comparator unit requires two input voltages, a positive and 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 negative channel inputs are connected to a pair of MUXs, 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 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 MUXs configured to the same input channel, with the negative channel input MUXs used to set the lower and upper window bounds.

2.2.3 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 it can 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.

2.2.4 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 introduces by this mode.

2.2.5 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 power consumption but decreases the latency between each comparison result by automatically performing a comparison on every cycle of the module's clock.

2.2.6 Input and Output Events

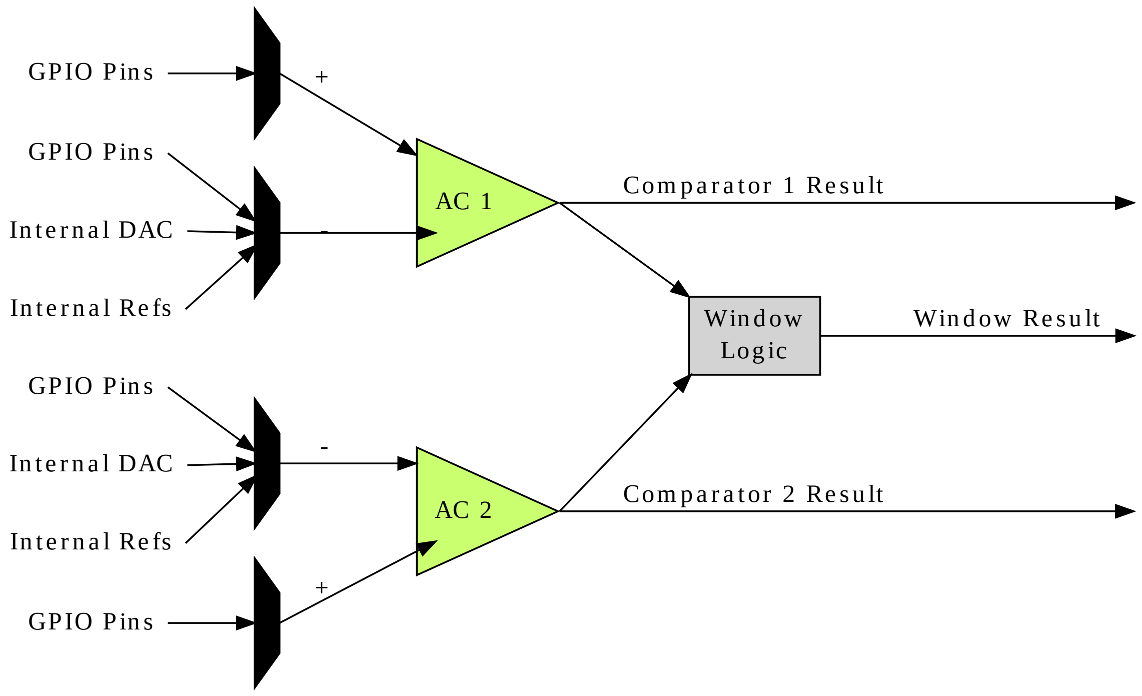
Each comparator unit is capable of being triggered by both software and hardware triggers. Hardware input events allow for other peripherals to automatically trigger a comparison on demand - for example, a timer output event could be used to trigger comparisons at a desired regular interval.

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.

2.2.7 Physical Connection

Physically, the modules are interconnected within the device as shown in Figure 2-1: Physical Connection.

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 AC

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

AC channel status flags

AC channel status flags, returned by ac_chan_get_status()

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

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 true, hysteresis mode is enabled on the comparator inputs.
enum ac_chan_filter	filter	Filtering mode for the comparator output, when the comparator is used in a supported mode.
enum ac_chan_interrupt_selection	interrupt_selection	Interrupt criteria for the comparator channel, to select the condition that will trigger a callback.
enum ac_chan_neg_mux	negative_input	Input multiplexer selection for the comparator's negative input pin.
enum ac_chan_output	output_mode	Output mode of the comparator, whether it should be available for internal use, or asynchronously/ synchronously linked to a GPIO pin.
enum ac_chan_pos_mux	positive_input	Input multiplexer selection for the comparator's positive input pin.
enum ac_chan_sample_mode	sample_mode	Sampling mode of the comparator channel.
uint8_t	vcc_scale_factor	Scaled \$\frac{V_{CC}\times \mbox{n}}{64}\$ VCC voltage division factor for the channel, when a comparator pin is connected to the VCC voltage scalar input. If the VCC voltage scalar is not selected as a comparator channel pin's input, this value will be ignored.

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
bool	run_in_standby[]	If true, the comparator pairs will continue to sample during sleep mode when triggered.
enum gclk_generator	source_generator	Source generator for AC GCLK.

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 true, an event will be generated when a comparator state changes.
bool	generate_event_on_window[]	If true, an event will be generated when a comparator window state changes.
bool	on_event_sample[]	If true, a comparator will be sampled each time an event is received.

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.

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

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 (1UL << 0)
```

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

Macro AC_WIN_STATUS_ABOVE

```
#define AC_WIN_STATUS_ABOVE (1UL << 1)
```

Window Comparator's input voltage is above the window

Macro AC_WIN_STATUS_INSIDE

```
#define AC_WIN_STATUS_INSIDE (1UL << 2)
```

Window Comparator's input voltage is inside the window

Macro AC_WIN_STATUS_BELOW

```
#define AC_WIN_STATUS_BELOW (1UL << 3)
```

Window Comparator's input voltage is below the window

Macro AC_WIN_STATUS_INTERRUPT_SET

```
#define AC_WIN_STATUS_INTERRUPT_SET (1UL << 4)
```

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().

AC channel status flags

AC channel status flags, returned by ac_chan_get_status()

Macro AC_CHAN_STATUS_UNKNOWN

```
#define AC_CHAN_STATUS_UNKNOWN (1UL << 0)
```

Unknown output state; the comparator channel was not ready.

Macro AC_CHAN_STATUS_NEG_ABOVE_POS

```
#define AC_CHAN_STATUS_NEG_ABOVE_POS (1UL << 1)
```

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 (1UL << 2)
```

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 (1UL << 3)
```

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

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, resetting the hardware module registers 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
true	if the module has completed synchronization
false	if the module synchronization is ongoing

Function ac_get_config_defaults()

Initializes an Analog Comparator configuration structure to defaults.

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

Initializes a given Analog Comparator 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:

- All comparator pairs disabled during sleep mode
- 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 and starts 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)
```

Stops 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 here 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 here 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

Channel Configuration and Initialization

Function ac_chan_get_config_defaults()

Initializes an Analog Comparator channel configuration structure to defaults.

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

Initializes a given Analog Comparator 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:

- Continuous sampling mode
- Majority of 5 sample output filter
- Hysteresis enabled on the input pins
- Internal comparator output mode
- Comparator pin multiplexer 0 selected as the positive input
- Scaled VCC voltage selected as the negative input
- VCC voltage scaler set for a division factor of 2 (

$$\frac{V_{CC} \times 32}{64} \tag{2-1}$$

)
- 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 and starts 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

Data direction	Parameter name	Description
[in]	channel	Comparator channel to disable

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

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

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 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]	module_inst	Software instance for the Analog Comparator peripheral
[in]	win_channel	Window channel to clear

2.6.5 Enumeration Definitions

AC channel status flags

AC channel status flags, returned by ac_chan_get_status()

Enum ac_callback

Enum for possible callback types for the AC module

Table 2-31. Members

Enum value	Description
AC_CALLBACK_COMPARATOR_0	Callback for comparator 0

Enum value	Description
AC_CALLBACK_COMPARATOR_1	Callback for comparator 1
AC_CALLBACK_WINDOW_0	Callback for window 0

Enum ac_chan_channel

Enum for the possible comparator channels.

Table 2-32. Members

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

Enum ac_chan_sample_mode

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

Table 2-33. 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.

Enum ac_chan_pos_mux

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

Table 2-34. 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.

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.
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 VCC 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	Negative comparator input is connected to the channel's internal DAC channel 0 output.

Enum ac_chan_filter

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

Table 2-36. 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.

Enum ac_chan_output

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

Table 2-37. 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.

Enum ac_win_channel

Enum for the possible window comparator channels.

Table 2-38. 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)

Enum ac_chan_interrupt_selection

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

Table 2-39. 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_COM	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 ac_chan_cleare_status().

Enum ac_win_interrupt_selection

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

Table 2-40. 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

Acronym	Description
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 D20 Analog Comparator Driver (AC). 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 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 (i.e. "Single Shot" mode)
- One comparator channel connected to input MUX pin 0 and compared to a scaled VCC/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 VCC 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.

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 = MUX_PA04B_AC_AIN0;
    system_pinmux_pin_set_config(PIN_PA04B_AC_AIN0, &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.

   Note Device instance structures should **never** go out of scope when in use.

   ```
   static struct ac_module ac_instance;
   ```

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.

Note The voltage scalar formula is documented here.

```
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;
```

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 = MUX_PA04B_AC_AIN0;
system_pinmux_pin_set_config(PIN_PA04B_AC_AIN0, &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);
```

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_STATE_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 (i.e. "Single Shot" mode)
- One comparator channel connected to input MUX pin 0 and compared to a scaled VCC/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 VCC 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.

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 = MUX_PA04B_AC_AIN0;
    system_pinmux_pin_set_config(PIN_PA04B_AC_AIN0, &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.

Note Device instance structures should **never** go out of scope when in use.

```
static struct ac_module ac_instance;
```

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 here.

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 = MUX_PA04B_AC_AIN0;
system_pinmux_pin_set_config(PIN_PA04B_AC_AIN0, &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 callback_status = false;
```

17. Let the callback function set the calback_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.

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

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

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));
        ac_chan_trigger_single_shot(&ac_instance, AC_COMPARATOR_CHANNEL);
        callback_status = false;
    }
}
```

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

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

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

3. SAM D20 Analog to Digital Converter Driver (ADC)

This driver for SAM D20 devices 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 API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripherals are used by this module:

- ADC (Analog to Digital Converter)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for ADC
- 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 500k samples per second (ksps).

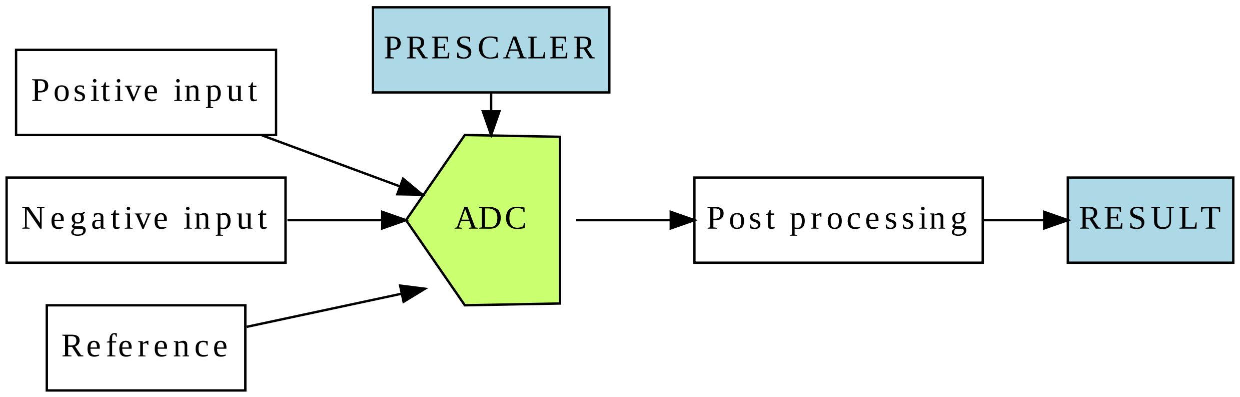
The ADC has a compare function for accurate monitoring of user defined thresholds with minimum software intervention required. The ADC may be configured for 8-, 10- or 12-bit result, reducing the conversion time from 2.0µs for 12-bit to 1.4µs for 8-bit result. ADC conversion results are provided left 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.

A simplified block diagram of the ADC can be seen in Figure 3-1: Module Overview.

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-bit, 10-bit 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-bits to an effective 13, 14, 15 or 16-bits. 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: Effective ADC conversion speed using oversampling.

Table 3-1. Effective ADC conversion speed using oversampling

Resolution	Effective conversion rate
13-bits	Conversion rate divided by 4
14-bits	Conversion rate divided by 16
15-bits	Conversion rate divided by 64
16-bits	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:

$$t_{SAMPLE} = (sample\_length + 1) \times \frac{ADC_{CLK}}{2} \tag{3-1}$$

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-bits 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: Effective ADC resolution from various hardware averaging modes.

Table 3-2. Effective ADC resolution from various hardware averaging modes

Number of Samples	Final Result
1	12-bits
2	13-bits
4	14-bits
8	15-bits
16	16-bits
32	16-bits
64	16-bits
128	16-bits
256	16-bits
512	16-bits
1024	16-bits

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:

$$ADC_{RESULT} = (VALUE_{CONV} + CORR_{OFFSET}) \times CORR_{GAIN} \qquad (3\text{-}2)$$

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.

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 compares 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.

3.3 Special Considerations

An integrated analog temperature sensor is available for use with the ADC. The bandgap voltage, as well as the scaled IO 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 ADC

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

Type adc_callback_t

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

Type of the callback functions

3.6.2 Structure Definitions

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 adc_accumulate_samples	accumulate_samples	Number of ADC samples to accumulate when using the ADC_RESOLUTION_CUSTOM mode
enum adc_clock_prescaler	clock_prescaler	Clock prescaler
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
struct adc_correction_config	correction	Gain and offset correction configuration structure
bool	differential_mode	Enables differential mode if true
enum adc_divide_result	divide_result	Division ration when using the ADC_RESOLUTION_CUSTOM mode
enum adc_event_action	event_action	Event action to take on incoming event
bool	freerunning	Enables free running mode if true
enum adc_gain_factor	gain_factor	Gain factor
bool	left_adjust	Left adjusted result
enum adc_negative_input	negative_input	Negative MUX input
struct adc_pin_scan_config	pin_scan	Pin scan configuration structure
enum adc_positive_input	positive_input	Positive MUX input
enum adc_reference	reference	Voltage reference
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	Enables ADC in standby sleep mode if true
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:

Type	Name	Description
		Sample time = (sample_length+1) * (ADCclk / 2)
struct adc_window_config	window	Window monitor configuration structure

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 gain_correction and offset_correction 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 <= gain_correction < 2. Valid gain_correction values ranges from 0b010000000000 to 0b111111111111.
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.

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	generate_event_on_conversion_don	Enable event generation on conversion done
bool	generate_event_on_window_monitor	Enable event generation on window monitor

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.

Struct adc_pin_scan_config

Pin scan configuration structure. Part of the adc_config struct and will be initialized by adc_get_config_defaults .

Table 3-6. Members

Type	Name	Description
uint8_t	inputs_to_scan	Number of input pins to scan in pin scan mode. A value below 2 will disable pin scan mode.
uint8_t	offset_start_scan	Offset (relative to selected positive input) of the first input pin to be used in pin scan mode.

Struct adc_window_config

Window monitor configuration structure.

Table 3-7. Members

Type	Name	Description
int32_t	window_lower_value	Lower window value
enum adc_window_mode	window_mode	Selected window mode
int32_t	window_upper_value	Upper window value

3.6.3 Macro Definitions

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 (1UL << 0)
```

ADC result ready

Macro ADC_STATUS_WINDOW

```
#define ADC_STATUS_WINDOW (1UL << 1)
```

Window monitor match

Macro ADC_STATUS_OVERRUN

```
#define ADC_STATUS_OVERRUN (1UL << 2)
```

ADC result overwritten before read

3.6.4 Function Definitions

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 by for the interrupt handler to call it when the condition for the callback is met.

Table 3-8. 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 3-9. 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 3-10. 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 3-11. 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 3-12. 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 3-13. Return Values

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

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

Table 3-14. 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 3-15. 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 3-16. Parameters

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

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 3-17. Parameters

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

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 3-18. Parameters

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

Returns Status of the initialization procedure

Table 3-19. 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
- 1V from internal bandgap reference
- Div 4 clock prescaler
- 12 bit resolution
- Window monitor disabled
- No gain
- Positive input on ADC PIN 0
- Negative input on ADC PIN 1
- 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
- Factory gain/offset correction
- No added sampling time
- Pin scan mode disabled

Table 3-20. Parameters

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

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 3-21. Parameters

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

Returns Bitmask of ADC_STATUS_* flags

Table 3-22. Return Values

Return value	Description
ADC_STATUS_RESULT_READY	ADC Result is ready to be read
ADC_STATUS_WINDOW	ADC has detected a value inside the set window range
ADC_STATUS_OVERRUN	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 3-23. Parameters

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

Enable, disable and reset ADC module, start conversion and read result

Function adc_is_syncing()

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

```
bool adc_is_syncing(
  struct adc_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 3-24. Parameters

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

Returns Synchronization status of the underlying hardware module(s).

Table 3-25. Return Values

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

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.

Table 3-26. 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 3-27. 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 3-28. 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 here for a list of events this module supports.

Note Events cannot be altered while the module is enabled.

Table 3-29. 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 here for a list of events this module supports.

Note Events cannot be altered while the module is enabled.

Table 3-30. 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 3-31. 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 3-32. 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 3-33. Return Values

Return value	Description
STATUS_OK	The result was retrieved successfully
STATUS_BUSY	A conversion result was not ready

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 restart 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 3-34. 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 3-35. 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_gain()

Sets ADC gain factor.

```
void adc_set_gain(
  struct adc_module *const module_inst,
  const enum adc_gain_factor gain_factor)
```

Sets the ADC gain factor to a specified gain setting.

Table 3-36. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	gain_factor	Gain factor value to set

Function adc_set_pin_scan_mode()

Sets the ADC pin scan mode.

```
enum status_code adc_set_pin_scan_mode(
  struct adc_module *const module_inst,
  uint8_t inputs_to_scan,
  const uint8_t start_offset)
```

Configures the pin scan mode of the ADC module. In pin scan mode, the first conversion will start at the configured positive input + start_offset. When a conversion is done, a conversion will start on the next input, until `inputs_to_scan` number of conversions are made.

Table 3-37. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	inputs_to_scan	Number of input pins to perform a conversion on (must be two or more)
[in]	start_offset	Offset of first pin to scan (relative to configured positive input)

Returns Status of the pin scan configuration set request.

Table 3-38. Return Values

Return value	Description
STATUS_OK	Pin scan mode has been set successfully
STATUS_ERR_INVALID_ARG	Number of input pins to scan or offset has an invalid value

Function adc_disable_pin_scan_mode()

Disables pin scan mode.

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

Disables pin scan mode. The next conversion will be made on only one pin (the configured positive input pin).

Table 3-39. Parameters

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

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 3-40. 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 3-41. Parameters

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

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 3-42. 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 3-43. Parameters

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

Data direction	Parameter name	Description
[in]	interrupt	Interrupt to disable

3.6.5 Enumeration Definitions

Enum adc_accumulate_samples

Enum for the possible numbers of ADC samples to accumulate. This setting is only used when the ADC_RESOLUTION_CUSTOM [64] resolution setting is used.

Table 3-44. 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 265 samples
ADC_ACCUMULATE_SAMPLES_512	Average 512 samples
ADC_ACCUMULATE_SAMPLES_1024	Average 1024 samples

Enum adc_callback

Callback types for ADC callback driver

Table 3-45. 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

Enum adc_clock_prescaler

Enum for the possible clock prescaler values for the ADC.

Table 3-46. Members

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

Enum value	Description
ADC_CLOCK_PRESCALER_DIV128	ADC clock division factor 128
ADC_CLOCK_PRESCALER_DIV256	ADC clock division factor 256
ADC_CLOCK_PRESCALER_DIV512	ADC clock division factor 512

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 [64] resolution setting is used.

Table 3-47. 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

Enum adc_event_action

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

Table 3-48. 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

Enum adc_gain_factor

Enum for the possible gain factor values for the ADC.

Table 3-49. Members

Enum value	Description
ADC_GAIN_FACTOR_1X	1x gain
ADC_GAIN_FACTOR_2X	2x gain
ADC_GAIN_FACTOR_4X	4x gain
ADC_GAIN_FACTOR_8X	8x gain
ADC_GAIN_FACTOR_16X	16x gain
ADC_GAIN_FACTOR_DIV2	1/2x gain

Enum adc_interrupt_flag

Enum for the possible ADC interrupt flags

Table 3-50. 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

Enum adc_job_type

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

Table 3-51. Members

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

Enum adc_negative_input

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

Table 3-52. 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_PIN8	ADC8 pin
ADC_NEGATIVE_INPUT_PIN9	ADC9 pin
ADC_NEGATIVE_INPUT_PIN10	ADC10 pin
ADC_NEGATIVE_INPUT_PIN11	ADC11 pin
ADC_NEGATIVE_INPUT_PIN12	ADC12 pin
ADC_NEGATIVE_INPUT_PIN13	ADC13 pin
ADC_NEGATIVE_INPUT_PIN14	ADC14 pin
ADC_NEGATIVE_INPUT_PIN15	ADC15 pin
ADC_NEGATIVE_INPUT_PIN16	ADC16 pin
ADC_NEGATIVE_INPUT_PIN17	ADC17 pin
ADC_NEGATIVE_INPUT_PIN18	ADC18 pin
ADC_NEGATIVE_INPUT_PIN19	ADC19 pin
ADC_NEGATIVE_INPUT_PIN20	ADC20 pin
ADC_NEGATIVE_INPUT_PIN21	ADC21 pin

Enum value	Description
ADC_NEGATIVE_INPUT_PIN22	ADC22 pin
ADC_NEGATIVE_INPUT_PIN23	ADC23 pin
ADC_NEGATIVE_INPUT_GND	Internal ground
ADC_NEGATIVE_INPUT_IOGND	I/O ground

Enum adc_oversampling_and_decimation

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

Table 3-53. Members

Enum value	Description
ADC_OVERSAMPLING_AND_DECIMATION_DISABLE	Don't use oversampling and decimation mode
ADC_OVERSAMPLING_AND_DECIMATION_1BIT	1 bit resolution increase
ADC_OVERSAMPLING_AND_DECIMATION_2BIT	2 bits resolution increase
ADC_OVERSAMPLING_AND_DECIMATION_3BIT	3 bits resolution increase
ADC_OVERSAMPLING_AND_DECIMATION_4BIT	4 bits resolution increase

Enum adc_positive_input

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

Table 3-54. Members

Enum value	Description
ADC_POSITIVE_INPUT_PIN0	ADC0 pin
ADC_POSITIVE_INPUT_PIN1	ADC1 pin
ADC_POSITIVE_INPUT_PIN2	ADC2 pin
ADC_POSITIVE_INPUT_PIN3	ADC3 pin
ADC_POSITIVE_INPUT_PIN4	ADC4 pin
ADC_POSITIVE_INPUT_PIN5	ADC5 pin
ADC_POSITIVE_INPUT_PIN6	ADC6 pin
ADC_POSITIVE_INPUT_PIN7	ADC7 pin
ADC_POSITIVE_INPUT_PIN8	ADC8 pin
ADC_POSITIVE_INPUT_PIN9	ADC9 pin
ADC_POSITIVE_INPUT_PIN10	ADC10 pin
ADC_POSITIVE_INPUT_PIN11	ADC11 pin
ADC_POSITIVE_INPUT_PIN12	ADC12 pin
ADC_POSITIVE_INPUT_PIN13	ADC13 pin
ADC_POSITIVE_INPUT_PIN14	ADC14 pin
ADC_POSITIVE_INPUT_PIN15	ADC15 pin
ADC_POSITIVE_INPUT_PIN16	ADC16 pin
ADC_POSITIVE_INPUT_PIN17	ADC17 pin
ADC_POSITIVE_INPUT_PIN18	ADC18 pin
ADC_POSITIVE_INPUT_PIN19	ADC19 pin

Enum value	Description
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_SCALEDCOREVCC	1/4 scaled core supply
ADC_POSITIVE_INPUT_SCALEDIOVCC	1/4 scaled I/O supply
ADC_POSITIVE_INPUT_DAC	DAC input

Enum adc_reference

Enum for the possible reference voltages for the ADC.

Table 3-55. Members

Enum value	Description
ADC_REFERENCE_INT1V	1.0V voltage reference
ADC_REFERENCE_INTVCC0	1/1.48 VCC reference
ADC_REFERENCE_INTVCC1	1/2 VCC (only for internal Vcc > 2.1v)
ADC_REFERENCE_AREFA	External reference A
ADC_REFERENCE_AREFB	External reference B

Enum adc_resolution

Enum for the possible resolution values for the ADC.

Table 3-56. 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
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

Enum adc_window_mode

Enum for the possible window monitor modes for the ADC.

Table 3-57. 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

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 D20 Analog to Digital Converter Driver (ADC). 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 ADC - Basic
- Quick Start Guide for ADC - Callback

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 0
- Negative input on ADC PIN 1
- 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

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);

    adc_init(&adc_instance, ADC, &config_adc);

    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 it is in use.

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

```
struct adc_module adc_instance;
```

2. Configure the ADC module.

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

   ```
   struct adc_config config_adc;
   ```

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

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

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

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

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

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 be 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 compete.

The ADC will be set up as follows:

- VCC / 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

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(
        const 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);

    config_adc.gain_factor     = ADC_GAIN_FACTOR_DIV2;
    config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV8;
    config_adc.reference       = ADC_REFERENCE_INTVCC1;
    config_adc.positive_input  = ADC_POSITIVE_INPUT_PIN4;
    config_adc.resolution      = ADC_RESOLUTION_12BIT;

    adc_init(&adc_instance, ADC, &config_adc);

    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 it is in use.

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

```
struct adc_module adc_instance;
```

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

4. Configure the ADC module.

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

   ```
   struct adc_config config_adc;
   ```

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

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

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

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

   ```
   config_adc.gain_factor     = ADC_GAIN_FACTOR_DIV2;
   config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV8;
   config_adc.reference       = ADC_REFERENCE_INTVCC1;
   config_adc.positive_input  = ADC_POSITIVE_INPUT_PIN4;
   config_adc.resolution      = ADC_RESOLUTION_12BIT;
   ```

   d. 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

   a. 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);
```

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

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

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 */
}
```

4. SAM D20 Brown Out Detector Driver (BOD)

This driver for SAM D20 devices 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 peripherals are used by this module:

- SYSCTRL (System Control)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for BOD
- Examples
- API Overview

4.1 Prerequisites

There are no prerequisites for this module.

4.2 Module Overview

The SAM D20 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.

4.3 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.

4.4 Extra Information for BOD

For extra information see Extra Information for BOD Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

4.5 Examples

For a list of examples related to this driver, see Examples for BOD Driver.

4.6 API Overview

4.6.1 Structure Definitions

Struct bod_config

Configuration structure for a BOD module.

Table 4-1. Members

Type	Name	Description
enum bod_action	action	Action to perform when a low power detection is made.
bool	hysteresis	If true, enables detection hysteresis.
uint8_t	level	BOD level to trigger at (see electrical section of device datasheet).
enum bod_mode	mode	Sampling configuration mode for the BOD.
enum bod_prescale	prescaler	Input sampler clock prescaler factor, to reduce the 1KHz clock from the ULP32K to lower the sampling rate of the BOD.
bool	run_in_standby	If true, the BOD is kept enabled and sampled during device sleep.

4.6.2 Function Definitions

Configuration and Initialization

Function bod_get_config_defaults()

Get default BOD configuration.

```
void bod_get_config_defaults(
  struct bod_config *const conf)
```

The default BOD configuration is:

- Clock prescaler set to divide the input clock by 2
- Continuous mode
- Reset on BOD detect
- Hysteresis enabled
- BOD level 0x12
- BOD kept enabled during device sleep

Table 4-2. Parameters

Data direction	Parameter name	Description
[out]	conf	BOD configuration struct to set to default settings

Function bod_set_config()

Configure a Brown Out Detector module.

```
enum status_code bod_set_config(
  const enum bod bod_id,
  struct bod_config *const conf)
```

Configures a given BOD module with the settings stored in the given configuration structure.

Table 4-3. Parameters

Data direction	Parameter name	Description
[in]	bod_id	BOD module to configure
[in]	conf	Configuration settings to use for the specified BOD

Table 4-4. 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 bod_enable()

Enables a configured BOD module.

```
enum status_code bod_enable(
  const enum bod bod_id)
```

Enables the specified BOD module that has been previously configured.

Table 4-5. Parameters

Data direction	Parameter name	Description
[in]	bod_id	BOD module to enable

Returns Error code indicating the status of the enable operation.

Table 4-6. Return Values

Return value	Description
STATUS_OK	If the BOD was successfully enabled
STATUS_ERR_INVALID_ARG	An invalid BOD was supplied

Function bod_disable()

Disables an enabled BOD module.

```
enum status_code bod_disable(
  const enum bod bod_id)
```

Disables the specified BOD module that was previously enabled.

Table 4-7. Parameters

Data direction	Parameter name	Description
[in]	bod_id	BOD module to disable

Returns Error code indicating the status of the disable operation.

Table 4-8. Return Values

Return value	Description
STATUS_OK	If the BOD was successfully disabled
STATUS_ERR_INVALID_ARG	An invalid BOD was supplied

Function bod_is_detected()

Checks if a specified BOD low voltage detection has occurred.

```
bool bod_is_detected(
  const enum bod bod_id)
```

Determines if a specified BOD has detected a voltage lower than its configured threshold.

Table 4-9. Parameters

Data direction	Parameter name	Description
[in]	bod_id	BOD module to check

Returns Detection status of the specified BOD.

Table 4-10. Return Values

Return value	Description
true	If the BOD has detected a low voltage condition
false	If the BOD has not detected a low voltage condition

Function bod_clear_detected()

Clears the low voltage detection state of a specified BOD.

```
void bod_clear_detected(
  const enum bod bod_id)
```

Clears the low voltage condition of a specified BOD module, so that new low voltage conditions can be detected.

Table 4-11. Parameters

Data direction	Parameter name	Description
[in]	bod_id	BOD module to clear

4.6.3 Enumeration Definitions

Enum bod

List of possible BOD controllers within the device.

Table 4-12. Members

Enum value	Description
BOD_BOD12	BOD12 Internal core voltage.
BOD_BOD33	BOD33 External I/O voltage,

Enum bod_action

List of possible BOD actions when a BOD module detects a brown-out condition.

Table 4-13. Members

Enum value	Description
BOD_ACTION_NONE	A BOD detect will do nothing, and the BOD state must be polled.
BOD_ACTION_RESET	A BOD detect will reset the device.
BOD_ACTION_INTERRUPT	A BOD detect will fire an interrupt.

Enum bod_mode

List of possible BOD module voltage sampling modes.

Table 4-14. Members

Enum value	Description
BOD_MODE_CONTINUOUS	BOD will sample the supply line continuously.
BOD_MODE_SAMPLED	BOD will use the BOD sampling clock (1kHz) to sample the supply line.

Enum bod_prescale

List of possible BOD controller prescaler values, to reduce the sampling speed of a BOD to lower the power consumption.

Table 4-15. Members

Enum value	Description
BOD_PRESCALE_DIV_2	Divide input prescaler clock by 2
BOD_PRESCALE_DIV_4	Divide input prescaler clock by 4
BOD_PRESCALE_DIV_8	Divide input prescaler clock by 8
BOD_PRESCALE_DIV_16	Divide input prescaler clock by 16
BOD_PRESCALE_DIV_32	Divide input prescaler clock by 32
BOD_PRESCALE_DIV_64	Divide input prescaler clock by 64
BOD_PRESCALE_DIV_128	Divide input prescaler clock by 128
BOD_PRESCALE_DIV_256	Divide input prescaler clock by 256
BOD_PRESCALE_DIV_512	Divide input prescaler clock by 512

Enum value	Description
BOD_PRESCALE_DIV_1024	Divide input prescaler clock by 1024
BOD_PRESCALE_DIV_2048	Divide input prescaler clock by 2048
BOD_PRESCALE_DIV_4096	Divide input prescaler clock by 4096
BOD_PRESCALE_DIV_8192	Divide input prescaler clock by 8192
BOD_PRESCALE_DIV_16384	Divide input prescaler clock by 16384
BOD_PRESCALE_DIV_32768	Divide input prescaler clock by 32768
BOD_PRESCALE_DIV_65536	Divide input prescaler clock by 65536

4.7 Extra Information for BOD Driver

4.7.1 Acronyms

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

Acronym	Definition
BOD	Brownout detector

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 BOD Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Brown Out Detector Driver (BOD). 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 BOD - Basic

4.8.1 Quick Start Guide for BOD - Basic

In this use case, the BOD33 will be configured with the following settings:

- Continuous sampling mode
- Prescaler setting of 2

- Reset action on low voltage detect

Quick Start

Prerequisites

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

Code

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

```
void configure_bod33(void)
{
    struct bod_config config_bod33;
    bod_get_config_defaults(&config_bod33);

    bod_set_config(BOD_BOD33, &config_bod33);

    bod_enable(BOD_BOD33);
}
```

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

```
configure_bod33();
```

Workflow

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

```
struct bod_config config_bod33;
```

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

Note

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

```
bod_get_config_defaults(&config_bod33);
```

3. Configure the BOD module with the desired settings.

```
bod_set_config(BOD_BOD33, &config_bod33);
```

4. Enable the BOD module so that it will monitor the power supply voltage.

```
bod_enable(BOD_BOD33);
```

Use Case

Code

Copy-paste the following code to your user application:

```
while (true) {

}
```

Workflow

1. Enter an infinite loop so that the BOD can continue to monitor the supply voltage level.

   ```
   while (true) {

   }
   ```

5. SAM D20 Clock Management Driver (CLOCK)

This driver for SAM D20 devices 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)
- SYSCTRL (Clock Source Control)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for System Clock
- Examples
- API Overview

5.1 Prerequisites

There are no prerequisites for this module.

5.2 Module Overview

The SAM D20 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 functionality within the device.

5.2.1 Clock Sources

The SAM D20 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.

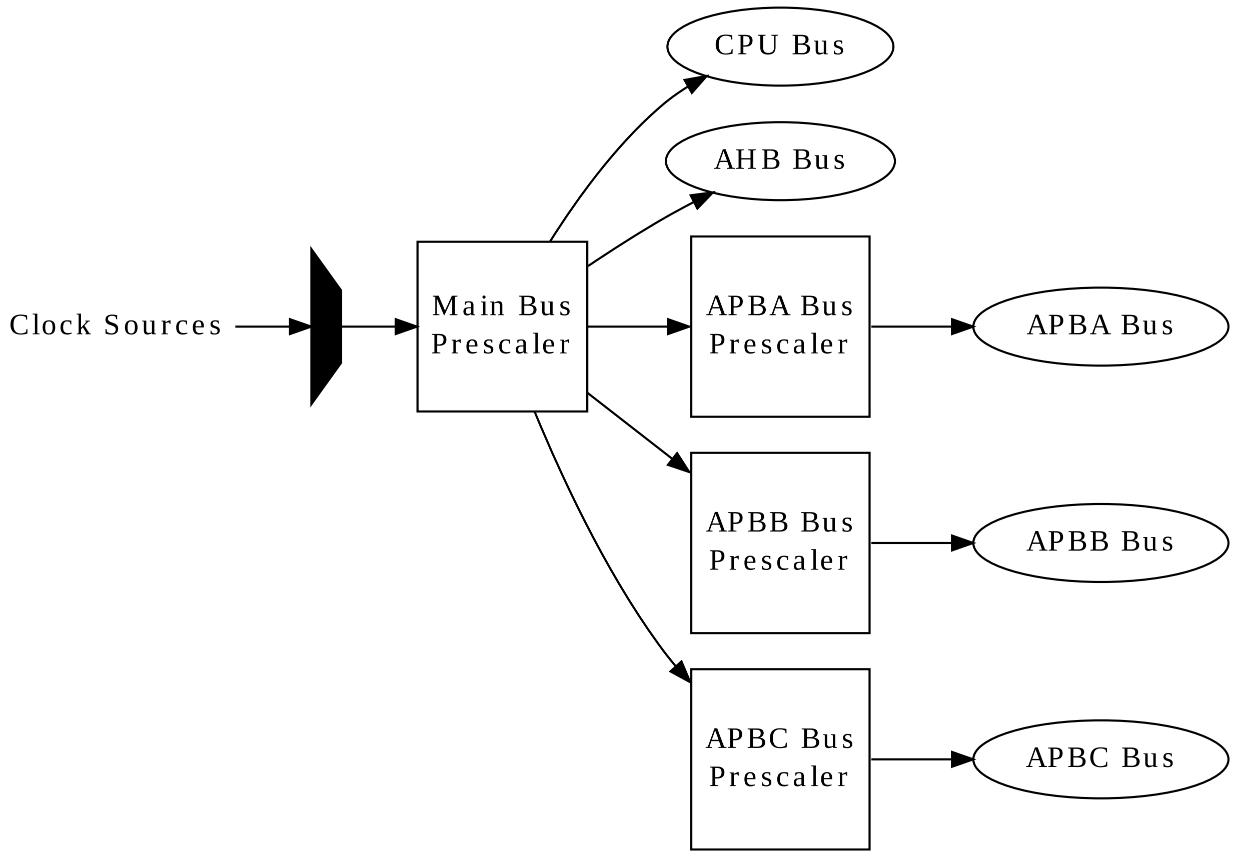
Possible clock source modules include internal R/C oscillators, internal DFLL modules, as well as external crystal oscillators and/or clock inputs.

5.2.2 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), however the APBx buses may have additional prescaler division ratios set to give each peripheral bus a different clock speed.

The general main clock tree for the CPU and associated buses is shown in Figure 5-1: CPU / Bus Clocks.

Figure 5-1. CPU / Bus Clocks



5.2.3 Clock Masking

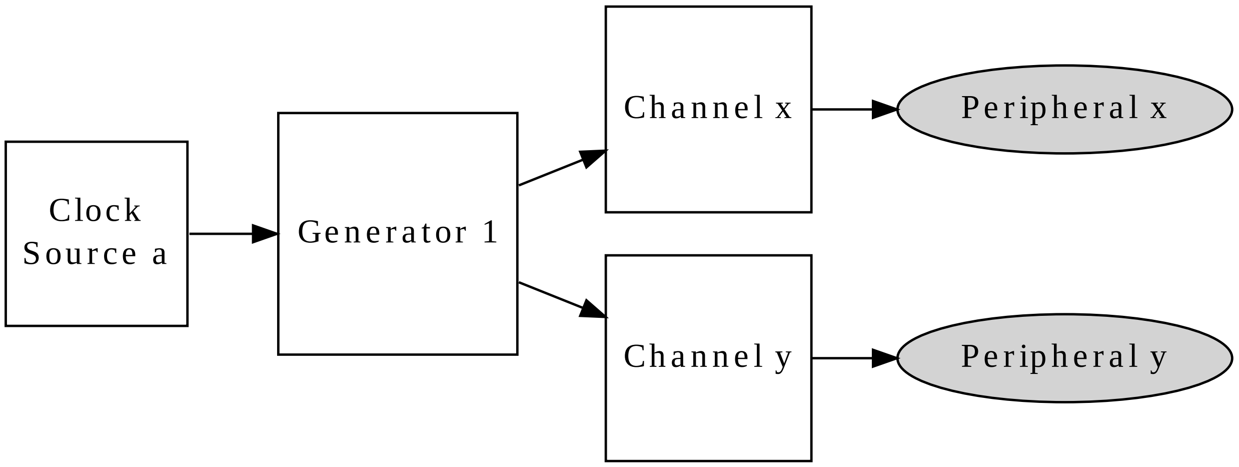
To save power, the input clock to one or more peripherals on the AHB and APBx busses can be masked away - when masked, no clock is passed into the module. Disabling of clocks of unused modules will prevent all access to the masked module, but will reduce the overall device power consumption.

5.2.4 Generic Clocks

Within the SAM D20 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.

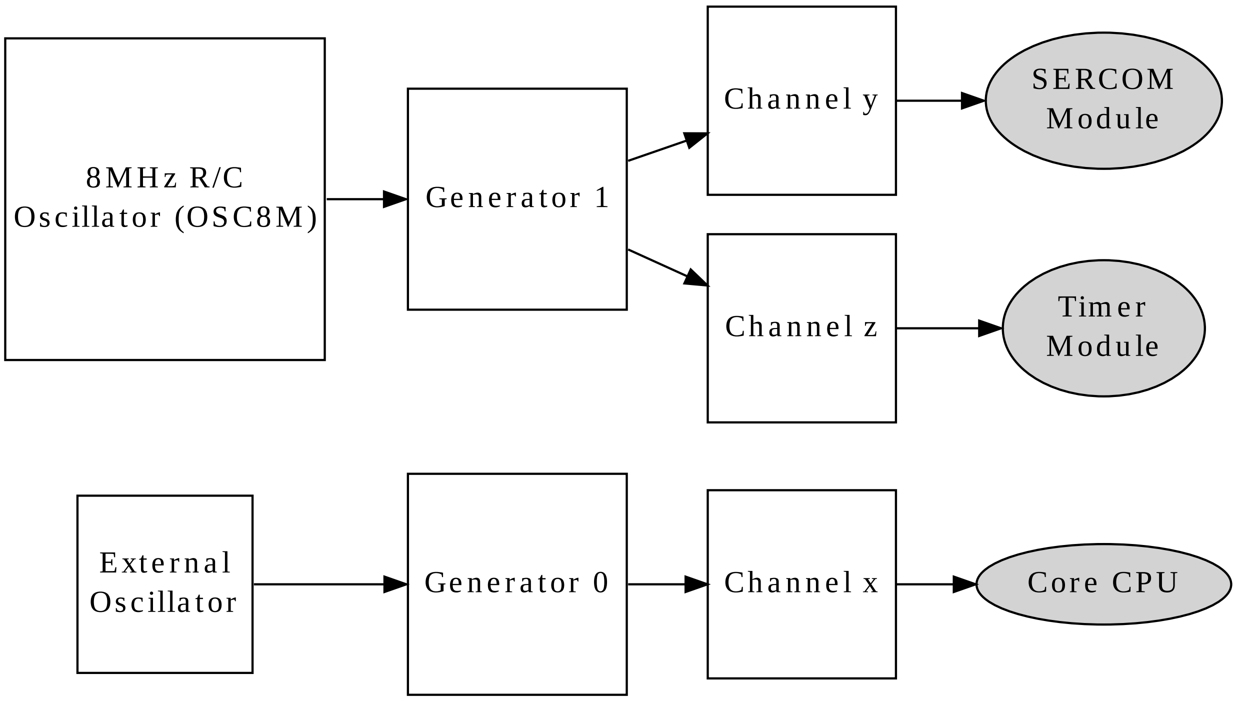
Figure 5-2. Generic Clocks



Clock Chain Example

An example setup of a complete clock chain within the device is shown in Figure 5-3: Clock Chain Example.

Figure 5-3. Clock Chain Example



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.

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.

5.3 Special Considerations

There are no special considerations for this module.

5.4 Extra Information for System Clock

For extra information see Extra Information for SYSTEM CLOCK Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

5.5 Examples

For a list of examples related to this driver, see Examples for System Clock Driver.

5.6 API Overview

5.6.1 Structure Definitions

Struct system_clock_source_dfll_config

DFLL oscillator configuration structure.

Table 5-1. Members

Type	Name	Description
enum system_clock_dfll_chill_cycle	chill_cycle	Enable Chill Cycle
uint8_t	coarse_max_step	Coarse adjustment max step size (Closed loop mode)
uint8_t	coarse_value	Coarse calibration value (Open loop mode)
uint8_t	fine_max_step	Fine adjustment max step size (Closed loop mode)
uint8_t	fine_value	Fine calibration value (Open loop mode)
enum system_clock_dfll_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_dfll_quick_lock	quick_lock	Enable Quick Lock
bool	run_in_standby	Keep the DFLL enabled in standby sleep mode
enum system_clock_dfll_stable_tracking	stable_tracking	DFLL tracking after fine lock

Type	Name	Description
enum system_clock_dfll_wakeup_lock	wakeup_lock	DFLL lock state on wakeup

Struct system_clock_source_osc32k_config

Internal 32KHz (nominal) oscillator configuration structure.

Table 5-2. Members

Type	Name	Description
bool	enable_1khz_output	Enable 1kHz output
bool	enable_32khz_output	Enable 32kHz output
bool	on_demand	Run On Demand. If this is set the OSC32K won't run until requested by a peripheral
bool	run_in_standby	Keep the OSC32K enabled in standby sleep mode
enum system_osc32k_startup	startup_time	Startup time

Struct system_clock_source_osc8m_config

Internal 8MHz (nominal) oscillator configuration structure.

Table 5-3. Members

Type	Name	Description
bool	on_demand	Run On Demand. If this is set the OSC8M won't run until requested by a peripheral
enum system_osc8m_div	prescaler	
bool	run_in_standby	Keep the OSC8M enabled in standby sleep mode

Struct system_clock_source_xosc32k_config

External 32KHz oscillator clock configuration structure.

Table 5-4. Members

Type	Name	Description
bool	auto_gain_control	Enable automatic amplitude control
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
bool	on_demand	Run On Demand. If this is set the XOSC32K won't run until requested by a peripheral

Type	Name	Description
bool	run_in_standby	Keep the XOSC32K enabled in standby sleep mode
enum system_xosc32k_startup	startup_time	Crystal oscillator start-up time

Struct system_clock_source_xosc_config

External oscillator clock configuration structure.

Table 5-5. Members

Type	Name	Description
bool	auto_gain_control	Enable automatic amplitude gain control
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

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 5-6. Members

Type	Name	Description
enum gclk_generator	source_generator	Generic Clock Generator source channel.
bool	write_lock	If true the clock configuration will be locked until the device is reset.

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 5-7. Members

Type	Name	Description
uint32_t	division_factor	Integer division factor of the clock output compared to the input.
bool	high_when_disabled	If true, the generator output level is high when disabled.
bool	output_enable	If true, enables GCLK generator clock output to a GPIO pin.

Type	Name	Description
bool	run_in_standby	If true, the clock is kept enabled during device standby mode.
uint8_t	source_clock	Source clock input channel index.

5.6.2 Function Definitions

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)

Table 5-8. 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 5-9. Parameters

Data direction	Parameter name	Description
[in]	config	External oscillator configuration structure containing the new config

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 enabled
- 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)

Table 5-10. Parameters

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

Function system_clock_source_xosc32k_set_config()

Configure the XOSC32K 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 5-11. Parameters

Data direction	Parameter name	Description
[in]	config	XOSC32K configuration structure containing the new config

Internal 32KHz Oscillator management

Function system_clock_source_osc32k_get_config_defaults()

Retrieve the default configuration for OSC32K.

```
void system_clock_source_osc32k_get_config_defaults(
  struct system_clock_source_osc32k_config *const config)
```

Fills a configuration structure with the default configuration for an internal 32KHz oscillator module:

- 1KHz clock output enabled
- 32KHz clock output enabled
- Don't run in STANDBY sleep mode
- Run only when requested by peripheral (on demand)

Table 5-12. Parameters

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

Function system_clock_source_osc32k_set_config()

Configure the internal OSC32K oscillator clock source.

```
void system_clock_source_osc32k_set_config(
  struct system_clock_source_osc32k_config *const config)
```

Configures the 32KHz (nominal) internal RC oscillator with the given configuration settings.

Table 5-13. Parameters

Data direction	Parameter name	Description
[in]	config	OSC32K configuration structure containing the new config

Internal 8MHz Oscillator management

Function system_clock_source_osc8m_get_config_defaults()

Retrieve the default configuration for OSC8M.

```
void system_clock_source_osc8m_get_config_defaults(
  struct system_clock_source_osc8m_config *const config)
```

Fills a configuration structure with the default configuration for an internal 8MHz (nominal) oscillator module:

- Clock output frequency divided by a factor of 8
- Don't run in STANDBY sleep mode
- Run only when requested by peripheral (on demand)

Table 5-14. Parameters

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

Function system_clock_source_osc8m_set_config()

Configure the internal OSC8M oscillator clock source.

```
void system_clock_source_osc8m_set_config(
  struct system_clock_source_osc8m_config *const config)
```

Configures the 8MHz (nominal) internal RC oscillator with the given configuration settings.

Table 5-15. Parameters

Data direction	Parameter name	Description
[in]	config	OSC8M configuration structure containing the new config

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 5-16. 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 5-17. Parameters

Data direction	Parameter name	Description
[in]	config	DFLL configuration structure containing the new config

Clock source management

Function system_clock_source_write_calibration()

```
enum status_code system_clock_source_write_calibration(
  const enum system_clock_source system_clock_source,
  const uint16_t calibration_value,
  const uint8_t freq_range)
```

Function system_clock_source_enable()

```
enum status_code system_clock_source_enable(
  const enum system_clock_source system_clock_source)
```

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 5-18. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to disable

Table 5-19. 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 5-20. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to check if ready

Returns Ready state of the given clock source.

Table 5-21. 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_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 5-22. Parameters

Data direction	Parameter name	Description
[in]	clock_source	Clock source to get the frequency of

Returns Frequency of the given clock source, in Hz

Main clock management

Function system_main_clock_set_failure_detect()

Enable or disable the main clock failure detection.

```
void system_main_clock_set_failure_detect(
  const bool enable)
```

This mechanism allows switching automatically the main clock to the safe RCSYS clock, when the main clock source is considered off.

This may happen for instance when an external crystal is selected as the clock source of the main clock and the crystal dies. The mechanism is to detect, during a RCSYS period, at least one rising edge of the main clock. If no rising edge is seen the clock is considered failed. As soon as the detector is enabled, the clock failure detector

CFD) will monitor the divided main clock. When a clock failure is detected, the main clock automatically switches to the RCSYS clock and the CFD interrupt is generated if enabled.

Note The failure detect must be disabled if the system clock is the same or slower than 32kHz as it will believe the system clock has failed with a too-slow clock.

Table 5-23. Parameters

Data direction	Parameter name	Description
[in]	enable	Boolean `true` to enable, `false` to disable detection

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 5-24. Parameters

Data direction	Parameter name	Description
[in]	divider	CPU clock divider to set

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_apb_clock_set_divider()

Set APBx clock divider.

```
enum status_code system_apb_clock_set_divider(
  const enum system_clock_apb_bus bus,
  const enum system_main_clock_div divider)
```

Set the clock divider used on the main clock to provide the clock for the given APBx bus.

Table 5-25. Parameters

Data direction	Parameter name	Description
[in]	divider	APBx bus divider to set

Data direction	Parameter name	Description
[in]	bus	APBx bus to set divider for

Returns Status of the clock division change operation.

Table 5-26. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid bus ID was given
STATUS_OK	The APBx clock was set successfully

Function system_apb_clock_get_hz()

Retrieves the current frequency of a ABPx.

```
uint32_t system_apb_clock_get_hz(
  const enum system_clock_apb_bus bus)
```

Retrieves the operating frequency of an APBx bus, obtained from the main generic clock and the set APBx bus divider.

Returns Current APBx bus frequency in Hz.

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 5-27. Parameters

Data direction	Parameter name	Description
[in]	ahb_mask	AHB clock mask to enable

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, 0 bits in the mask will be ignored.

Table 5-28. Parameters

Data direction	Parameter name	Description
[in]	ahb_mask	AHB clock mask to disable

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 5-29. Parameters

Data direction	Parameter name	Description
[in]	mask	APBx clock mask, a `SYSTEM_CLOCK_APB_APBx` constant from the device header files
[in]	bus	Bus to set clock mask bits for, a mask of `PM_APBxMASK_*` constants from the device header files

Returns Status indicating the result of the clock mask change operation.

Table 5-30. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid bus given
STATUS_OK	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 5-31. Parameters

Data direction	Parameter name	Description
[in]	mask	APBx clock mask, a `SYSTEM_CLOCK_APB_APBx`

Data direction	Parameter name	Description
		constant from the device header files
[in]	bus	Bus to clear clock mask bits for

Returns Status indicating the result of the clock mask change operation.

Table 5-32. Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	Invalid bus ID was given.
STATUS_OK	The clock mask was changed successfully.

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.

Handler for the CPU Hard Fault interrupt, fired if an illegal access was attempted to a memory address.

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 5-33. Parameters

Data direction	Parameter name	Description
[in]	wait_states	Number of wait states to use for internal flash

Generic Clock management

Function system_gclk_is_syncing()

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

```
bool system_gclk_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 5-34. Return Values

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

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.

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 as follows:

- Clock is generated undivided from the source frequency
- Clock generator output is low when the generator is disabled
- The input clock is sourced from input clock channel 0
- Clock will be disabled during sleep
- The clock output will not be routed to a physical GPIO pin

Table 5-35. 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.

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 5-36. 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 5-37. 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 5-38. Parameters

Data direction	Parameter name	Description
[in]	generator	Generic Clock Generator index to disable

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:

- Clock is sourced from the Generic Clock Generator channel 0
- Clock configuration will not be write-locked when set

Table 5-39. 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 5-40. 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 5-41. 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 5-42. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock channel to disable

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 5-43. 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 5-44. Parameters

Data direction	Parameter name	Description
[in]	channel	Generic Clock Channel index

Returns The frequency of the generic clock channel, in Hz.

5.6.3 Enumeration Definitions

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 5-45. 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.

Enum system_clock_apb_bus

Available bus clock domains on the APB bus.

Table 5-46. Members

Enum value	Description
SYSTEM_CLOCK_APB_APBA	Peripheral bus A on the APB bus.

Enum value	Description
SYSTEM_CLOCK_APB_APBB	Peripheral bus B on the APB bus.
SYSTEM_CLOCK_APB_APBC	Peripheral bus C on the APB bus.

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 5-47. 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

Enum system_clock_dfll_loop_mode

Available operating modes of the DFLL clock source module,

Table 5-48. 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

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 5-49. 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

Enum system_clock_dfll_stable_tracking

DFLL fine tracking behavior modes after a lock has been acquired.

Table 5-50. Members

Enum value	Description
SYSTEM_CLOCK_DFLL_STABLE_TRACKING_TRACK	Keep tracking after the DFLL has gotten a fine lock
SYSTEM_CLOCK_DFLL_STABLE_TRACKING_FIX_AF	Stop tracking after the DFLL has gotten a fine lock

Enum system_clock_dfll_wakeup_lock

DFLL lock behavior modes on device wake-up from sleep.

Table 5-51. 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

Enum system_clock_external

Available external clock source types.

Table 5-52. 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

Enum system_clock_source

Clock sources available to the GCLK generators

Table 5-53. Members

Enum value	Description
SYSTEM_CLOCK_SOURCE_OSC8M	Internal 8MHz RC oscillator
SYSTEM_CLOCK_SOURCE_OSC32K	Internal 32kHz 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

Enum system_main_clock_div

Available division ratios for the CPU and APB/AHB bus clocks.

Table 5-54. 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

Enum system_osc32k_startup

Available internal 32KHz oscillator start-up times, as a number of internal OSC32K clock cycles.

Table 5-55. Members

Enum value	Description
SYSTEM_OSC32K_STARTUP_0	Wait 0 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_2	Wait 2 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_4	Wait 4 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_8	Wait 8 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_16	Wait 16 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_32	Wait 32 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_64	Wait 64 clock cycles until the clock source is considered stable
SYSTEM_OSC32K_STARTUP_128	Wait 128 clock cycles until the clock source is considered stable

Enum system_osc8m_div

Available prescalers for the internal 8MHz (nominal) system clock.

Table 5-56. Members

Enum value	Description
SYSTEM_OSC8M_DIV_1	Do not divide the 8MHz RC oscillator output
SYSTEM_OSC8M_DIV_2	Divide the 8MHz RC oscillator output by 2
SYSTEM_OSC8M_DIV_4	Divide the 8MHz RC oscillator output by 4
SYSTEM_OSC8M_DIV_8	Divide the 8MHz RC oscillator output by 8

Enum system_xosc32k_startup

Available external 32KHz oscillator start-up times, as a number of external clock cycles.

Table 5-57. Members

Enum value	Description
SYSTEM_XOSC32K_STARTUP_0	Wait 0 clock cycles until the clock source is considered stable
SYSTEM_XOSC32K_STARTUP_32	Wait 32 clock cycles until the clock source is considered stable
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

Enum value	Description
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

Enum system_xosc_startup

Available external oscillator start-up times, as a number of external clock cycles.

Table 5-58. Members

Enum value	Description
SYSTEM_XOSC_STARTUP_1	Wait 1 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_2	Wait 2 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_4	Wait 4 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_8	Wait 8 clock cycles until the clock source is considered stable
SYSTEM_XOSC_STARTUP_16	Wait 16 clock cycles until the clock source is considered stable
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

5.7 Extra Information for SYSTEM CLOCK Driver

5.7.1 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
OSC32K	Internal 32KHz Oscillator
OSC8M	Internal 8MHz Oscillator
PLL	Phase Locked Loop
OSC	Oscillator
XOSC	External Oscillator
XOSC32K	External 32KHz Oscillator
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus

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 System Clock Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Clock Management Driver (CLOCK). 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 SYSTEM CLOCK - Basic
- Quick Start Guide for SYSTEM CLOCK - GCLK Configuration

5.8.1 Quick Start Guide for SYSTEM CLOCK - Basic

In this case we apply the following configuration:

- RC8MHz (internal 8MHz RC oscillator)

  - Divide by 4, giving a frequency of 2MHz
- DFLL (Digital frequency locked loop)
  - Open loop mode
  - 48MHz frequency
- CPU clock
  - Use the DFLL, configured to 48MHz

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);
}

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);
}
```

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.

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

```
system_clock_source_xosc32k_get_config_defaults(&config_ext32k);
```

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.

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

```
system_clock_source_dfll_get_config_defaults(&config_dfll);
```

7. Write the new configuration to the DFLL module.

```
system_clock_source_xosc32k_set_config(&config_ext32k);
```

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 */
}

/* 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 */
}

/* 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);
```

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. 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);
```

5.8.2 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.

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);

    gclock_gen_conf.source_clock    = SYSTEM_CLOCK_SOURCE_OSC8M;
    gclock_gen_conf.division_factor = 128;
    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;
    system_gclk_chan_set_config(TC0_GCLK_ID, &gclk_chan_conf);

    system_gclk_chan_enable(TC0_GCLK_ID);
}
```

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.

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

   ```
   system_gclk_gen_get_config_defaults(&gclock_gen_conf);
   ```

3. Adjust the configuration struct to request that the master clock source channel 0 be 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_OSC8M;
   gclock_gen_conf.division_factor = 128;
   ```

4. Configure the generator using the configuration structure.

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.

```
system_gclk_gen_set_config(GCLK_GENERATOR_1, &gclock_gen_conf);
```

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.

Note

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

```
system_gclk_chan_get_config_defaults(&gclk_chan_conf);
```

8. Adjust the configuration struct to request that the previously configured and enabled clock generator be used as the clock source for the channel.

```
gclk_chan_conf.source_generator = GCLK_GENERATOR_1;
```

9. Configure the channel using the configuration structure.

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.

```
system_gclk_chan_set_config(TC0_GCLK_ID, &gclk_chan_conf);
```

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(TC0_GCLK_ID);
```

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.

6. SAM D20 Digital-to-Analog Driver (DAC)

This driver for SAM D20 devices provides an interface for the conversion of digital values to analog voltage. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripherals are used by this module:

- DAC (Digital to Analog Converter)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for DAC
- Examples
- API Overview

6.1 Prerequisites

There are no prerequisites for this module.

6.2 Module Overview

The Digital-to-Analog converter converts a digital value to analog voltage. The SAM D20 DAC module has one channel with 10-bit resolution, and is capable of converting up to 350k samples per second (ksps).

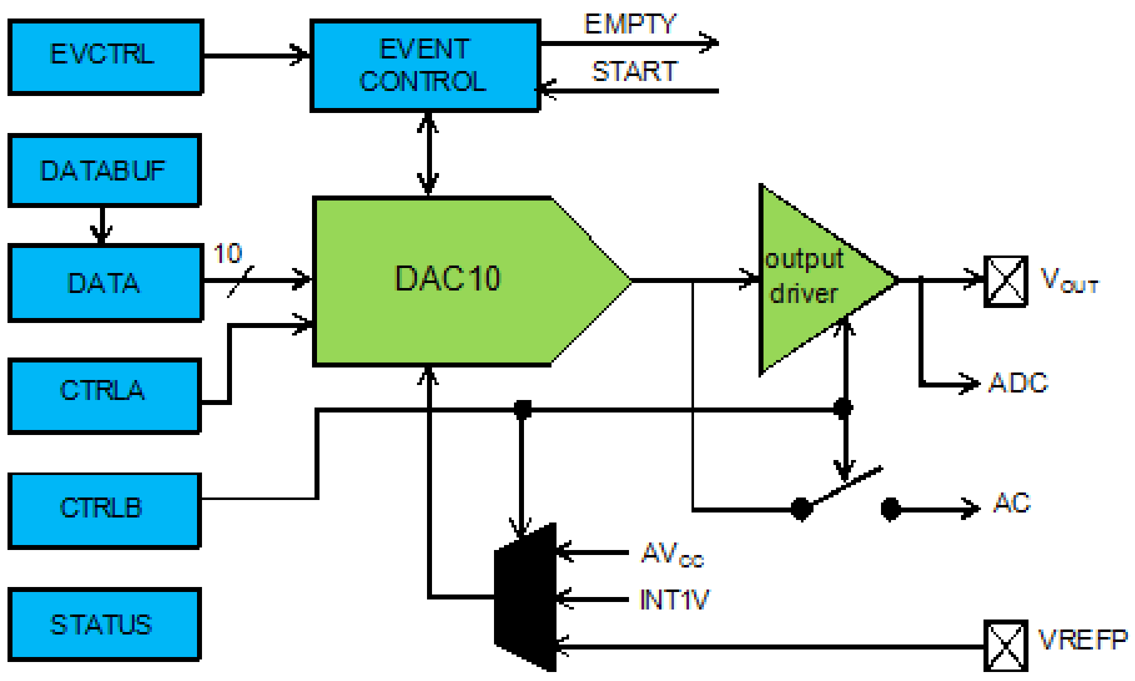
A common use of DAC is to generate audio signals by connecting the DAC output to a speaker, or to generate a reference voltage; either for an external circuit or an internal peripheral such as the Analog Comparator.

After being set up, the DAC will convert new digital values written to the conversion data register (DATA) to an analog value either on the VOUT pin of the device, or internally for use as an input to the AC, ADC and other analog modules.

Writing the DATA register will start a new conversion. It is also possible to trigger the conversion from the event system.

A simplified block diagram of the DAC can be seen in Figure 6-1: DAC Block Diagram.

Figure 6-1. DAC Block Diagram



6.2.1 Conversion Range

The conversion range is between GND and the selected voltage reference. Available voltage references are:

- AVCC voltage reference
- Internal 1V reference (INT1V)
- External voltage reference (AREF)

The output voltage from a DAC channel is given as:

$$V_{OUT} = \frac{DATA}{0x3FF} \times VREF \qquad (6\text{-}1)$$

6.2.2 Conversion

The digital value written to the conversion data register (DATA) will be converted to an analog value. Writing the DATA register will start a new conversion. It is also possible to write the conversion data to the DATABUF register, the writing of the DATA register can then be triggered from the event system, which will load the value from DATABUF to DATA.

6.2.3 Analog Output

The analog output value can be output to either the VOUT pin or internally, but not both at the same time.

External Output

The output buffer must be enabled in order to drive the DAC output to the VOUT pin. Due to the output buffer, the DAC has high drive strength, and is capable of driving both resistive and capacitive loads, as well as loads which combine both.

Internal Output

The analog value can be internally available for use as input to the AC or ADC modules.

6.2.4 Events

Events generation and event actions are configurable in the DAC. The DAC has one event line input and one event output: *Start Conversion* and *Data Buffer Empty*.

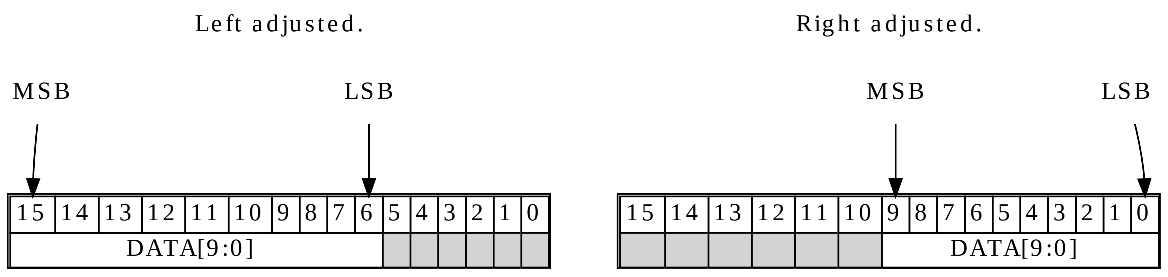
If the Start Conversion input event is enabled in the module configuration, an incoming event will load data from the data buffer to the data register and start a new conversion. This method synchronizes conversions with external events (such as those from a timer module) and ensures regular and fixed conversion intervals.

If the Data Buffer Empty output event is enabled in the module configuration, events will be generated when the DAC data buffer register becomes empty and new data can be loaded to the buffer.

6.2.5 Left and Right Adjusted Values

The 10-bit input value to the DAC is contained in a 16-bit register. This can be configured to be either left or right adjusted. In Figure 6-2: Left and Right Adjusted Values both options are shown, and the position of the most (MSB) and the least (LSB) significant bits are indicated. The unused bits should always be written to zero.

Figure 6-2. Left and Right Adjusted Values



6.2.6 Clock Sources

The clock for the DAC interface (CLK_DAC) is generated by the Power Manager. This clock is turned on by default, and can be enabled and disabled in the Power Manager.

Additionally, an asynchronous clock source (GCLK_DAC) is required. These clocks are normally disabled by default. The selected clock source must be enabled in the Power Manager before it can be used by the DAC. The DAC core operates asynchronously from the user interface and peripheral bus. As a consequence, the DAC needs two clock cycles of both CLK_DAC and GCLK_DAC to synchronize the values written to some of the control and data registers. The oscillator source for the GCLK_DAC clock is selected in the System Control Interface (SCIF).

6.3 Special Considerations

6.3.1 Output Driver

The DAC can only do conversions in Active or Idle modes. However, if the output buffer is enabled it will draw current even if the system is in sleep mode. Therefore, always make sure that the output buffer is not enabled when it is not needed, to ensure minimum power consumption.

6.3.2 Conversion Time

DAC conversion time is approximately 2.85us. The user must ensure that new data is not written to the DAC before the last conversion is complete. Conversions should be triggered by a periodic event from a Timer/Counter or another peripheral.

6.4 Extra Information for DAC

For extra information see Extra Information for DAC Driver. This includes:

- Acronyms
- Dependencies

- Errata
- Module History

6.5 Examples

For a list of examples related to this driver, see Examples for DAC Driver.

6.6 API Overview

6.6.1 Variable and Type Definitions

Callback configuration and initialization

Type dac_callback_t

```
typedef void(* dac_callback_t )(uint8_t channel)
```

Type definition for a DAC module callback function.

6.6.2 Structure Definitions

Struct dac_chan_config

Configuration for a DAC channel. This structure should be initialized by the dac_chan_get_config_defaults() function before being modified by the user application.

Struct dac_config

Configuration structure for a DAC instance. This structure should be initialized by the dac_get_config_defaults() function before being modified by the user application.

Table 6-1. Members

Type	Name	Description
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
bool	left_adjust	Left adjusted data
enum dac_output	output	Select DAC output
enum dac_reference	reference	Reference voltage
bool	run_in_standby	The DAC behaves as in normal mode when the chip enters STANDBY sleep mode

Struct dac_events

Event flags for the DAC module. This is used to enable and disable events via dac_enable_events() and dac_disable_events().

Table 6-2. Members

Type	Name	Description
bool	generate_event_on_buffer_empty	Enable event generation on data buffer empty
bool	on_event_start_conversion	Start a new DAC conversion

Struct dac_module

DAC 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.

6.6.3 Macro Definitions

DAC status flags

DAC status flags, returned by dac_get_status() and cleared by dac_clear_status().

Macro DAC_STATUS_CHANNEL_0_EMPTY

```
#define DAC_STATUS_CHANNEL_0_EMPTY (1UL << 0)
```

Data Buffer Empty Channel 0 - Set when data is transferred from DATABUF to DATA by a start conversion event and DATABUF is ready for new data.

Macro DAC_STATUS_CHANNEL_0_UNDERRUN

```
#define DAC_STATUS_CHANNEL_0_UNDERRUN (1UL << 1)
```

Under-run Channel 0 - Set when a start conversion event occurs when DATABUF is empty.

6.6.4 Function Definitions

Callback configuration and initialization

Function dac_register_callback()

Registers an asynchronous callback function with the driver.

```
enum status_code dac_register_callback(
  struct dac_module *const module,
  const dac_callback_t callback,
  const enum dac_callback type)
```

Registers an asynchronous callback with the DAC driver, fired when a callback condition occurs.

Table 6-3. Parameters

Data direction	Parameter name	Description
[inout]	dac_module	Pointer to the DAC software instance struct
[in]	callback	Pointer to the callback function to register
[in]	type	Type of callback function to register

Returns Status of the registration operation.

Table 6-4. 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_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode.

Function dac_unregister_callback()

Unregisters an asynchronous callback function with the driver.

```
enum status_code dac_unregister_callback(
  struct dac_module *const module,
  const enum dac_callback type)
```

Unregisters an asynchronous callback with the DAC driver, removing it from the internal callback registration table.

Table 6-5. Parameters

Data direction	Parameter name	Description
[inout]	dac_module	Pointer to the DAC software instance struct
[in]	type	Type of callback function to unregister

Returns Status of the de-registration operation.

Table 6-6. 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_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode.

Callback enabling and disabling (Channel)

Function dac_chan_enable_callback()

Enables asynchronous callback generation for a given channel and type.

```
enum status_code dac_chan_enable_callback(
  struct dac_module *const module,
  const uint32_t channel,
  const enum dac_callback type)
```

Enables asynchronous callbacks for a given logical DAC channel and type. This must be called before a DAC channel will generate callback events.

Table 6-7. Parameters

Data direction	Parameter name	Description
[inout]	dac_module	Pointer to the DAC software instance struct
[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 6-8. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully.
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied.
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode.

Function dac_chan_disable_callback()

Disables asynchronous callback generation for a given channel and type.

```
enum status_code dac_chan_disable_callback(
  struct dac_module *const module,
  const uint32_t channel,
  const enum dac_callback type)
```

Disables asynchronous callbacks for a given logical DAC channel and type.

Table 6-9. Parameters

Data direction	Parameter name	Description
[inout]	dac_module	Pointer to the DAC software instance struct
[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 6-10. Return Values

Return value	Description
STATUS_OK	The callback was disabled successfully.

Return value	Description
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied.
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode.

Configuration and Initialization

Function dac_is_syncing()

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

```
bool dac_is_syncing(
  struct dac_module *const dev_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 6-11. Parameters

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

Returns Synchronization status of the underlying hardware module(s).

Table 6-12. Return Values

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

Function dac_get_config_defaults()

Initializes a DAC configuration structure to defaults.

```
void dac_get_config_defaults(
  struct dac_config *const config)
```

Initializes a given DAC 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:

- 1V from internal bandgap reference
- Drive the DAC output to the VOUT pin
- Right adjust data
- GCLK generator 0 (GCLK main) clock source

- The output buffer is disabled when the chip enters STANDBY sleep mode

Table 6-13. Parameters

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

Function dac_init()

Initialize the DAC device struct.

```
enum status_code dac_init(
  struct dac_module *const dev_inst,
  Dac *const module,
  struct dac_config *const config)
```

Use this function to initialize the Digital to Analog Converter. Resets the underlying hardware module and configures it.

Note The DAC channel must be configured separately.

Table 6-14. Parameters

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

Returns Status of initialization

Table 6-15. Return Values

Return value	Description
STATUS_OK	Module initiated correctly
STATUS_ERR_DENIED	If module is enabled
STATUS_BUSY	If module is busy resetting

Function dac_reset()

Resets the DAC module.

```
void dac_reset(
  struct dac_module *const dev_inst)
```

This function will reset the DAC module to its power on default values and disable it.

Table 6-16. Parameters

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

Function dac_enable()

Enable the DAC module.

```
void dac_enable(
  struct dac_module *const dev_inst)
```

Enables the DAC interface and the selected output.

Table 6-17. Parameters

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

Function dac_disable()

Disable the DAC module.

```
void dac_disable(
  struct dac_module *const dev_inst)
```

Disables the DAC interface and the output buffer.

Table 6-18. Parameters

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

Function dac_enable_events()

Enables a DAC event input or output.

```
void dac_enable_events(
  struct dac_module *const module_inst,
  struct dac_events *const events)
```

Enables one or more input or output events to or from the DAC module. See here for a list of events this module supports.

Note Events cannot be altered while the module is enabled.

Table 6-19. Parameters

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

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to enable

Function dac_disable_events()

Disables a DAC event input or output.

```
void dac_disable_events(
  struct dac_module *const module_inst,
  struct dac_events *const events)
```

Disables one or more input or output events to or from the DAC module. See here for a list of events this module supports.

Note Events cannot be altered while the module is enabled.

Table 6-20. Parameters

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

Configuration and Initialization (Channel)

Function dac_chan_get_config_defaults()

Initializes a DAC channel configuration structure to defaults.

```
void dac_chan_get_config_defaults(
  struct dac_chan_config *const config)
```

Initializes a given DAC channel 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:

- Start Conversion Event Input enabled
- Start Data Buffer Empty Event Output disabled

Table 6-21. Parameters

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

Function dac_chan_set_config()

Writes a DAC channel configuration to the hardware module.

```
void dac_chan_set_config(
  struct dac_module *const dev_inst,
  const enum dac_channel channel,
  struct dac_chan_config *const config)
```

Writes out a given channel configuration to the hardware module.

Note The DAC device instance structure must be initialized before calling this function.

Table 6-22. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Channel to configure
[in]	config	Pointer to the configuration struct

Function dac_chan_enable()

Enable a DAC channel.

```
void dac_chan_enable(
  struct dac_module *const dev_inst,
  enum dac_channel channel)
```

Enables the selected DAC channel.

Table 6-23. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Channel to enable

Function dac_chan_disable()

Disable a DAC channel.

```
void dac_chan_disable(
  struct dac_module *const dev_inst,
  enum dac_channel channel)
```

Disables the selected DAC channel.

Table 6-24. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Channel to disable

Function dac_chan_enable_output_buffer()

Enable the output buffer.

```
void dac_chan_enable_output_buffer(
  struct dac_module *const dev_inst,
  const enum dac_channel channel)
```

Enables the output buffer and drives the DAC output to the VOUT pin.

Table 6-25. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	DAC channel to alter

Function dac_chan_disable_output_buffer()

Disable the output buffer.

```
void dac_chan_disable_output_buffer(
  struct dac_module *const dev_inst,
  const enum dac_channel channel)
```

Disables the output buffer.

Note The output buffer(s) should be disabled when a channel's output is not currently needed, as it will draw current even if the system is in sleep mode.

Table 6-26. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	DAC channel to alter

Channel Data Management

Function dac_chan_write()

Write to the DAC.

```
enum status_code dac_chan_write(
  struct dac_module *const dev_inst,
  enum dac_channel channel,
  const uint16_t data)
```

This function writes to the DATA or DATABUF register. If the conversion is not event-triggered, the data will be written to the DATA register and the conversion will start. If the conversion is event-triggered, the data will be written to DATABUF and transferred to the DATA register and converted when a Start Conversion Event is issued. Conversion data must be right or left adjusted according to configuration settings.

Note To be event triggered, the enable_start_on_event must be enabled in the configuration.

Table 6-27. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	channel	DAC channel to write to
[in]	data	Conversion data

Returns Status of the operation

Table 6-28. Return Values

Return value	Description
STATUS_OK	If the data was written

Status Management

Function dac_get_status()

Retrieves the current module status.

```
uint32_t dac_get_status(
  struct dac_module *const module_inst)
```

Checks the status of the module and returns it as a bitmask of status flags

Table 6-29. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct

Returns Bitmask of status flags

Table 6-30. Return Values

Return value	Description
DAC_STATUS_CHANNEL_0_EMPTY	Data has been transferred from DATABUF to DATA by a start conversion event and DATABUF is ready for new data.
DAC_STATUS_CHANNEL_0_UNDERRUN	A start conversion event has occurred when DATABUF is empty

Function dac_clear_status()

Clears a module status flag.

```
void dac_clear_status(
  struct dac_module *const module_inst,
  uint32_t status_flags)
```

Clears the given status flag of the module.

Table 6-31. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	status_flags	Bit mask of status flags to clear

6.6.5 Enumeration Definitions

Callback configuration and initialization

Enum dac_callback

Enum for the possible callback types for the DAC module.

Table 6-32. Members

Enum value	Description
DAC_CALLBACK_DATA_EMPTY	Callback type for when a DAC channel data empty condition occurs (requires event triggered mode).
DAC_CALLBACK_DATA_UNDERRUN	Callback type for when a DAC channel data under-run condition occurs (requires event triggered mode).

Enum dac_channel

Enum for the DAC channel selection.

Table 6-33. Members

Enum value	Description
DAC_CHANNEL_0	DAC output channel 0.

Enum dac_output

Enum for the DAC output selection.

Table 6-34. Members

Enum value	Description
DAC_OUTPUT_EXTERNAL	DAC output to VOUT pin
DAC_OUTPUT_INTERNAL	DAC output as internal reference
DAC_OUTPUT_NONE	No output

Enum dac_reference

Enum for the possible reference voltages for the DAC.

Table 6-35. Members

Enum value	Description
DAC_REFERENCE_INT1V	1V from the internal band-gap reference.
DAC_REFERENCE_AVCC	Analog VCC as reference.

Enum value	Description
DAC_REFERENCE_AREF	External reference on AREF.

6.7 Extra Information for DAC Driver

6.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
ADC	Analog-to-Digital Converter
AC	Analog Comparator
DAC	Digital-to-Analog Converter
LSB	Least Significant Bit
MSB	Most Significant Bit

6.7.2 Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

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 DAC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Digital-to-Analog Driver (DAC). 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 DAC - Basic

6.8.1 Quick Start Guide for DAC - Basic

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

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

Quick Start

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;
```

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

```
void configure_dac(void)
{
    struct dac_config config_dac;
    dac_get_config_defaults(&config_dac);

    dac_init(&dac_instance, DAC, &config_dac);

    dac_enable(&dac_instance);
}

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);
}
```

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

```
configure_dac();
configure_dac_channel();
```

Workflow

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

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

```
struct dac_module dac_instance;
```

2. Configure the DAC module.

   a. 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;
   ```

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

Note

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

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

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

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

3. Configure the DAC channel.

   a. 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;
```

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

Note

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

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

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

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

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

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

Use Case

Code

Copy-paste the following code to your user application:

```
uint16_t i = 0;

while (1) {
    dac_chan_write(&dac_instance, DAC_CHANNEL_0, i);

    if (++i == 0x3FF) {
        i = 0;
    }
}
```

Workflow

1. Create a temporary variable to track the current DAC output value.

```
uint16_t i = 0;
```

2. Enter an infinite loop to continuously output new conversion values to the DAC.

```
while (1) {
```

3. Write the next conversion value to the DAC, so that it will be output on the device's DAC analog output pin.

```
dac_chan_write(&dac_instance, DAC_CHANNEL_0, i);
```

4. Increment and wrap the DAC output conversion value, so that a ramp pattern will be generated.

```
if (++i == 0x3FF) {
    i = 0;
}
```

7. SAM D20 Event System Driver

This driver for SAM D20 devices provides an interface for the configuration and management of the device's peripheral event channels and users within the device, including the enabling and disabling of peripheral source selection and synchronization of clock domains between various modules.

The following peripherals are used by this module:

- EVSYS (Event System Management)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for EVENTS
- Examples
- API Overview

7.1 Prerequisites

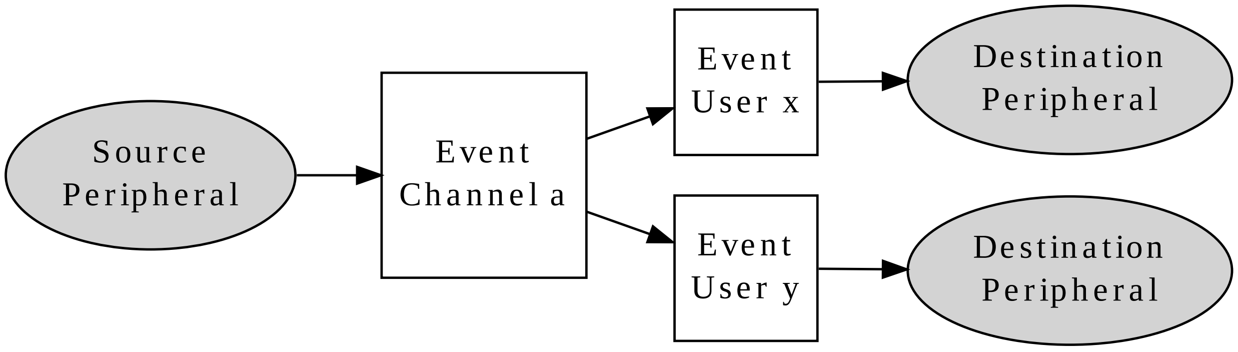
There are no prerequisites for this module.

7.2 Module Overview

Peripherals within the SAM D20 devices are capable of generating two types of actions in response to given stimulus; they can set a register flag for later intervention by the CPU (using interrupt or polling methods), or they can 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, and lowers the latency of the system response.

The event system is comprised of a number of freely configurable Event Channels, plus a number of fixed Event Users. Each Event Channel can be configured to select the input peripheral that will generate the events on the channel, 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 Channel 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 7-1: Module Overview.

Figure 7-1. 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 module to capture the current count value for later use.

7.2.1 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.

7.2.2 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.

7.2.3 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.

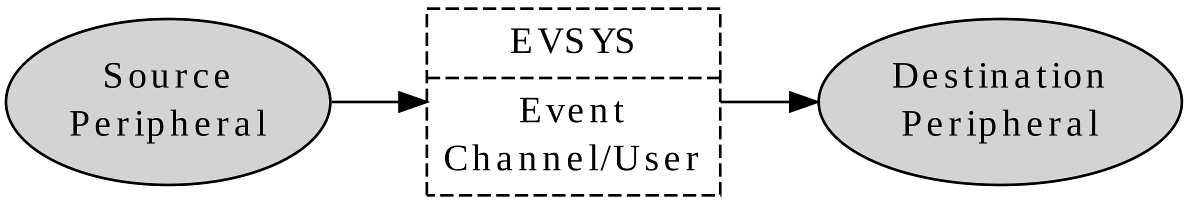
7.2.4 Path Selection

The event system in the SAM0 devices supports three signal path types from the event generator to event users: asynchronous, synchronous and re-synchronous events.

Asynchronous Paths

Asynchronous event paths allow for an asynchronous connection between the event generator and event user(s), 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 7-2: Asynchronous Paths.

Figure 7-2. Asynchronous Paths

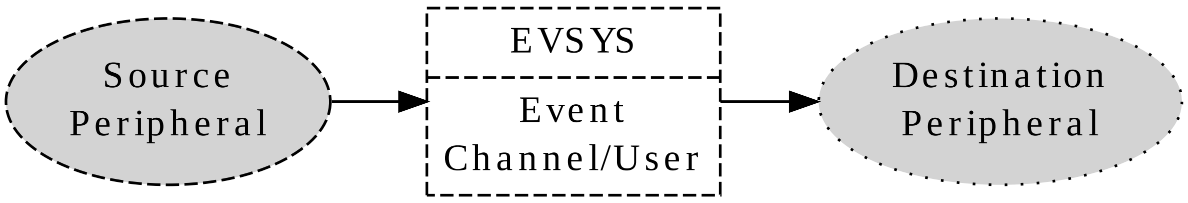


Note Identically shaped borders in the diagram indicate a shared generic clock channel

Synchronous Paths

Synchronous event paths can be used when the source and destination peripherals, as well as the generic clock to the event system itself, use different generic clock channels. This case introduces additional latency in the event propagation due to the addition of a synchronizer and edge detector on the input event signal, however this allows modules of different clocks to communicate events to one-another. The synchronous event chain is shown in Figure 7-3: Synchronous Paths.

Figure 7-3. Synchronous Paths

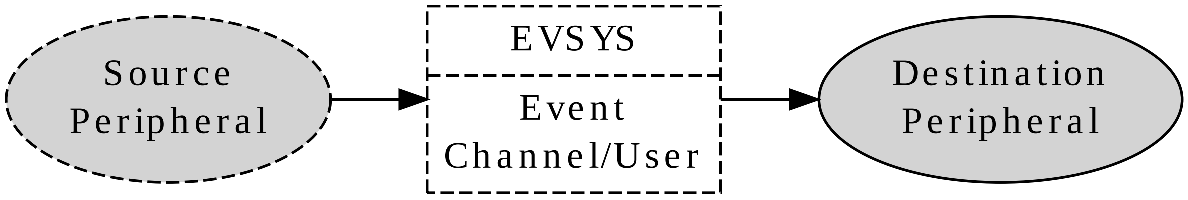


Note Identically shaped borders in the diagram indicate a shared generic clock channel

Re-synchronous Paths

Re-synchronous event paths are a special form of synchronous events, where the event users share the same generic clock channel as the event system module itself, but the event generator does not. This reduces latency by performing the synchronization across the event source and event user clock domains once within the event channel itself, rather than in each event user. The re-synchronous event chain is shown in Figure 7-4: Re-synchronous Paths.

Figure 7-4. Re-synchronous Paths

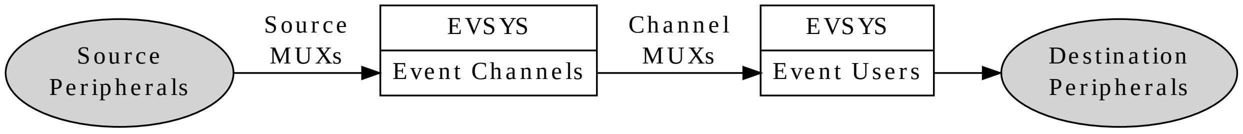


Note Identically shaped borders in the diagram indicate a shared generic clock channel

7.2.5 Physical Connection

Figure 7-5: Physical Connection shows how this module is interconnected within the device.

Figure 7-5. Physical Connection



7.3 Special Considerations

There are no special considerations for this module.

7.4 Extra Information for EVENTS

For extra information see Extra Information for EVENTS Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

7.5 Examples

For a list of examples related to this driver, see Examples for EVENTS Driver.

7.6 API Overview

7.6.1 Structure Definitions

Struct events_chan_config

Configuration structure for an Event System channel. This structure should be initialized by the events_chan_get_config_defaults() function before being modified by the user application.

Note Selecting a GLCK will only make take effect when EVENT_PATH_SYNCHRONOUS [139] and EVENT_PATH_RESYNCHRONOUS [139] paths are used.

Table 7-1. Members

Type	Name	Description
enum gclk_generator	clock_source	GCLK generator used to clock the specific event channel
enum events_edge	edge_detection	Edge detection for synchronous event channels, from events_edge.
uint8_t	generator_id	Event generator module that should be attached to the event channel, an EVSYS_ID_GEN_* constant from the device header files.
enum events_path	path	Path of the event system, from events_path.

Struct events_user_config

Configuration structure for an Event System subscriber multiplexer channel. This structure should be initialized by the events_user_get_config_defaults() function before being modified by the user application.

Table 7-2. Members

Type	Name	Description
enum events_channel	event_channel_id	Event channel ID that should be attached to the user MUX.

7.6.2 Function Definitions

Configuration and initialization

Function events_init()

Initializes the event driver.

```
void events_init(void)
```

Initializes the event driver ready for use. This resets the underlying hardware modules, clearing any existing event channel configuration(s).

Configuration and initialization (Event Channel)

Function events_is_syncing()

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

```
bool events_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 7-3. Return Values

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

Function events_chan_get_config_defaults()

Initializes an Event System configuration structure to defaults.

```
void events_chan_get_config_defaults(
  struct events_chan_config *const config)
```

Initializes a given Event System 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:

- Event channel uses asynchronous path between the source and destination
- Event channel is set not to use edge detection as the path is asynchronous and no intervention in the event system can take place
- Event channel is not connected to an Event Generator
- Event channel generic clock source is GLCK_GENERATOR_0
- Event channel generic clock does not run in standby mode

Table 7-4. Parameters

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

Function events_chan_set_config()

Writes an Event System channel configuration to the hardware module.

```
void events_chan_set_config(
  const enum events_channel event_channel,
  struct events_chan_config *const config)
```

Writes out a given configuration of a Event System channel configuration to the hardware module.

Precondition The user must be configured before the channel is configured, see events_user_set_config

Table 7-5. Parameters

Data direction	Parameter name	Description
[in]	event_channel	Event channel to configure

Data direction	Parameter name	Description
[in]	config	Configuration settings for the event channel

Configuration and initialization (Event User)

Function events_user_get_config_defaults()

Initializes an Event System user MUX configuration structure to defaults.

```
void events_user_get_config_defaults(
  struct events_user_config *const config)
```

Initializes a given Event System user MUX 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:

- User MUX input event is connected to source channel 0

Table 7-6. Parameters

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

Function events_user_set_config()

Writes an Event System user MUX configuration to the hardware module.

```
void events_user_set_config(
  const uint8_t user,
  struct events_user_config *const config)
```

Writes out a given configuration of a Event System user MUX configuration to the hardware module.

Table 7-7. Parameters

Data direction	Parameter name	Description
[in]	event_user	Event User MUX index to configure, a EVSYS_ID_USER_* constant from the device header files
[in]	config	Configuration settings for the event user MUX

Channel Control and Management

Function events_chan_is_ready()

Retrieves the busy status of an Event channel.

```
bool events_chan_is_ready(
  const enum events_channel event_channel)
```

Reads the status of the requested Event channel, to determine if the channel is currently busy.

Precondition The specified event channel must be configured and enabled.

Table 7-8. Parameters

Data direction	Parameter name	Description
[in]	event_channel	Event channel to check

Returns Status of the specified event channel.

Table 7-9. Return Values

Return value	Description
true	If the channel is ready to be used
false	If the channel is currently busy

Function events_user_is_ready()

Retrieves the channel status of the users subscribed to an Event channel.

```
bool events_user_is_ready(
  const enum events_channel event_channel)
```

Reads the status of the requested Event channel users, to determine if the users of the event channel are currently busy.

Precondition The specified event channel must be configured and enabled.

Table 7-10. Parameters

Data direction	Parameter name	Description
[in]	event_channel	Event channel to check

Returns Status of the specified event channel subscribers.

Table 7-11. Return Values

Return value	Description
true	If all channel subscribers are ready
false	If one or more channel subscribers are currently busy

Function events_chan_software_trigger()

Software triggers an event channel.

```
void events_chan_software_trigger(
  const enum events_channel event_channel)
```

Triggers an event channel via software, setting an event notification to the channel subscriber module(s) of the channel.

Precondition The specified event channel must be configured and enabled.

Table 7-12. Parameters

Data direction	Parameter name	Description
[in]	event_channel	Event channel to trigger

7.6.3 Enumeration Definitions

Enum events_channel

Enum containing the possible event channel selections.

Table 7-13. Members

Enum value	Description
EVENT_CHANNEL_0	Event channel 0
EVENT_CHANNEL_1	Event channel 1
EVENT_CHANNEL_2	Event channel 2
EVENT_CHANNEL_3	Event channel 3
EVENT_CHANNEL_4	Event channel 4
EVENT_CHANNEL_5	Event channel 5
EVENT_CHANNEL_6	Event channel 6
EVENT_CHANNEL_7	Event channel 7

Enum events_edge

Enum containing the possible event channel edge detection configurations, to select when the synchronous event triggers according to a particular trigger edge.

Note For asynchronous events, edge detection is not possible and selection of any value other than EVENT_EDGE_NONE [138] will have no effect. For synchronous events, a valid edge detection mode other than EVENT_EDGE_NONE [138] must be set for events to be generated.

Table 7-14. Members

Enum value	Description
EVENT_EDGE_NONE	Event channel disabled (or direct pass-through for asynchronous events).
EVENT_EDGE_RISING	Event channel triggers on rising edges.
EVENT_EDGE_FALLING	Event channel triggers on falling edges.
EVENT_EDGE_BOTH	Event channel triggers on both edges.

Enum events_path

Enum containing the possible event channel paths, to select between digital clock synchronization settings for each channel.

Table 7-15. Members

Enum value	Description
EVENT_PATH_SYNCHRONOUS	Event is synchronized to the digital clock.
EVENT_PATH_RESYNCHRONOUS	Event is re-synchronized between the source and destination digital clock domains.
EVENT_PATH_ASYNCHRONOUS	Event is asynchronous to the digital clock.

7.7 Extra Information for EVENTS Driver

7.7.1 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

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
Initial Release

7.8 Examples for EVENTS Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Event 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 QSGs can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for EVENTS - Basic

7.8.1 Quick Start Guide for EVENTS - Basic

In this use case, the EVENT module is configured for:

- One generator attached to event channel 0
- Synchronous event path with rising edge detection on the input
- One user attached to the configured event channel

This use case configures an event channel within the device, attaching it to a peripheral's event generator, and attaching a second peripheral's event user to the configured channel. The event channel is then software triggered.

Setup

Prerequisites

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

Code

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

```
#define EXAMPLE_EVENT_GENERATOR    0
#define EXAMPLE_EVENT_CHANNEL      EVENT_CHANNEL_0
#define EXAMPLE_EVENT_USER         0

void configure_event_channel(void)
{
    struct events_chan_config config_events_chan;
    events_chan_get_config_defaults(&config_events_chan);

    config_events_chan.generator_id   = EXAMPLE_EVENT_GENERATOR;
    config_events_chan.edge_detection = EVENT_EDGE_RISING;
    config_events_chan.path           = EVENT_PATH_SYNCHRONOUS;
    events_chan_set_config(EXAMPLE_EVENT_CHANNEL, &config_events_chan);
}

void configure_event_user(void)
{
    struct events_user_config config_events_user;
    events_user_get_config_defaults(&config_events_user);

    config_events_user.event_channel_id = EXAMPLE_EVENT_CHANNEL;
    events_user_set_config(EXAMPLE_EVENT_USER, &config_events_user);
}
```

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

```
events_init();

configure_event_user();
configure_event_channel();
```

Workflow

1. Create an event channel configuration struct, which can be filled out to adjust the configuration of a single event channel.

   ```
   struct events_chan_config config_events_chan;
   ```

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

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

   ```
   events_chan_get_config_defaults(&config_events_chan);
   ```

3. Adjust the configuration struct to request that the channel be attached to the specified event generator, that rising edges of the event signal be detected on the channel and that the synchronous event path be used.

   ```
   config_events_chan.generator_id   = EXAMPLE_EVENT_GENERATOR;
   config_events_chan.edge_detection = EVENT_EDGE_RISING;
   config_events_chan.path           = EVENT_PATH_SYNCHRONOUS;
   ```

4. Configure the channel using the configuration structure.

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.

```
events_chan_set_config(EXAMPLE_EVENT_CHANNEL, &config_events_chan);
```

5. Create an event user configuration struct, which can be filled out to adjust the configuration of a single event user.

```
struct events_user_config config_events_user;
```

6. Initialize the event user configuration struct with the module's default values.

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

```
events_user_get_config_defaults(&config_events_user);
```

7. Adjust the configuration struct to request that the previously configured event channel be used as the event source for the user.

```
config_events_user.event_channel_id = EXAMPLE_EVENT_CHANNEL;
```

8. Configure the event user using the configuration structure.

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.

```
events_user_set_config(EXAMPLE_EVENT_USER, &config_events_user);
```

Use Case

Code

Copy-paste the following code to your user application:

```
while (events_chan_is_ready(EXAMPLE_EVENT_CHANNEL) == false) {
    /* Wait for channel */
};

events_chan_software_trigger(EXAMPLE_EVENT_CHANNEL);

while (true) {
    /* Nothing to do */
}
```

Workflow

1. Wait for the even channel to become ready to accept a new event trigger.

```
while (events_chan_is_ready(EXAMPLE_EVENT_CHANNEL) == false) {
    /* Wait for channel */
};
```

2. Perform a software event trigger on the configured event channel.

```
events_chan_software_trigger(EXAMPLE_EVENT_CHANNEL);
```

8. SAM D20 External Interrupt Driver (EXTINT)

This driver for SAM D20 devices 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 peripherals are used by this module:

- EIC (External Interrupt Controller)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for EXTINT
- Examples
- API Overview

8.1 Prerequisites

There are no prerequisites for this module.

8.2 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.

8.2.1 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.

8.2.2 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. A 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.

8.2.3 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 8-1: Sampled input and resulting filtered output.

Table 8-1. 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

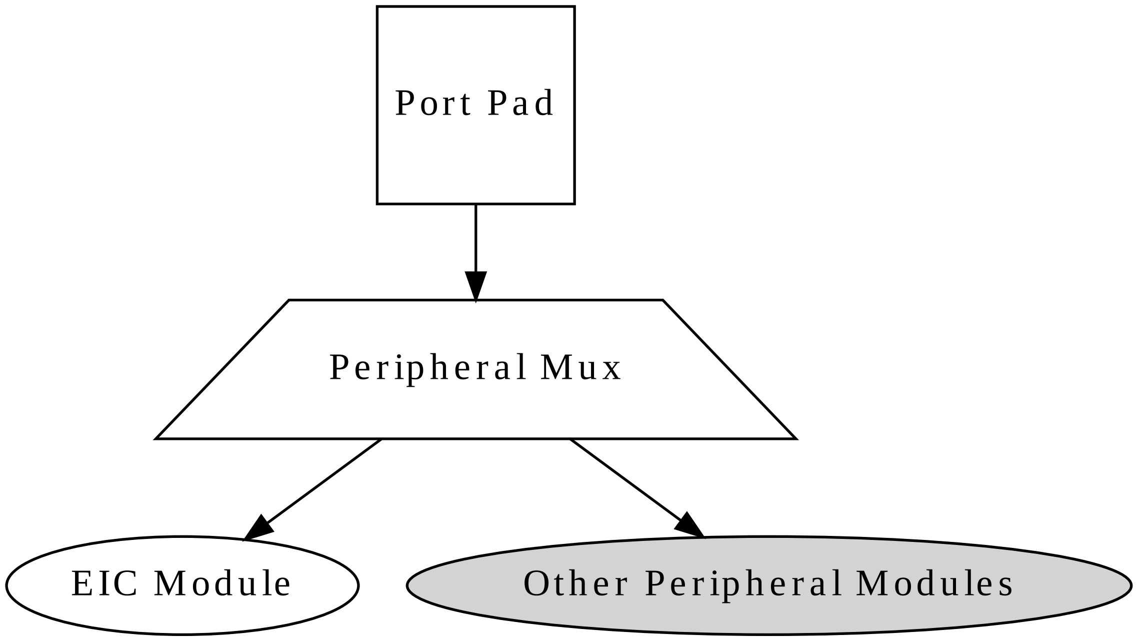
8.2.4 Events and Interrupts

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.

8.2.5 Physical Connection

Figure 8-1: Physical Connection shows how this module is interconnected within the device.

Figure 8-1. Physical Connection



8.3 Special Considerations

Not all devices support disabling of the NMI channel(s) detection mode - see your device datasheet.

8.4 Extra Information for EXTINT

For extra information see Extra Information for EXTINT Driver. This includes:

- Acronyms

- Dependencies
- Errata
- Module History

8.5 Examples

For a list of examples related to this driver, see Examples for EXTINT Driver.

8.6 API Overview

8.6.1 Variable and Type Definitions

Callback configuration and initialization

Type extint_callback_t

```
typedef void(* extint_callback_t )(uint32_t channel)
```

Type definition for an EXTINT module callback function.

8.6.2 Structure Definitions

Struct extint_chan_conf

Configuration structure for the edge detection mode of an external interrupt channel.

Table 8-2. Members

Type	Name	Description
enum extint_detect	detection_criteria	Edge detection mode to use.
bool	filter_input_signal	Filter the raw input signal to prevent noise from triggering an interrupt accidentally, using a 3 sample majority filter.
uint32_t	gpio_pin	GPIO pin the NMI should be connected to.
uint32_t	gpio_pin_mux	MUX position the GPIO pin should be configured to.
enum extint_pull	gpio_pin_pull	Internal pull to enable on the input pin.
bool	wake_if_sleeping	Wake up the device if the channel interrupt fires during sleep mode.

Struct extint_events

Event flags for the extint_enable_events() and extint_disable_events().

Table 8-3. Members

Type	Name	Description
bool	generate_event_on_detect[]	If true, an event will be generated when an external interrupt channel detection state changes.

Struct extint_nmi_conf

Configuration structure for the edge detection mode of an external interrupt NMI channel.

Table 8-4. Members

Type	Name	Description
enum extint_detect	detection_criteria	Edge detection mode to use. Not all devices support all possible detection modes for NMIs.
bool	filter_input_signal	Filter the raw input signal to prevent noise from triggering an interrupt accidentally, using a 3 sample majority filter.
uint32_t	gpio_pin	GPIO pin the NMI should be connected to.
uint32_t	gpio_pin_mux	MUX position the GPIO pin should be configured to.
enum extint_pull	gpio_pin_pull	Internal pull to enable on the input pin.

8.6.3 Macro Definitions

Macro EXTINT_CALLBACKS_MAX

```
#define EXTINT_CALLBACKS_MAX 10
```

Configuration option, setting the maximum number of callbacks which can be registered with the driver. This option may be overridden in the module configuration header file `conf_extint.h`.

8.6.4 Function Definitions

Configuration and initialization

Function extint_is_syncing()

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

```
bool extint_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 8-5. Return Values

Return value	Description
true	If the module has completed synchronization

Return value	Description
false	If the module synchronization is ongoing

Function extint_reset()

Resets and disables the External Interrupt driver.

```
void extint_reset(void)
```

Resets and disables the External Interrupt driver, resetting all hardware module registers to their power-on defaults.

Function extint_enable()

Enables the External Interrupt driver.

```
void extint_enable(void)
```

Enables EIC modules ready for use. This function must be called before attempting to use any NMI or standard external interrupt channel functions.

Function extint_disable()

Disables the External Interrupt driver.

```
void extint_disable(void)
```

Disables EIC modules that were previously started via a call to extint_enable().

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 8-6. 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 8-7. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to disable

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 8-8. 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 8-9. Parameters

Data direction	Parameter name	Description
[in]	channel	External Interrupt channel to configure
[in]	config	Configuration settings for the channel

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

Table 8-10. 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 8-11. 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 8-12. Return Values

Return value	Description
STATUS_OK	Configuration succeeded
STATUS_ERR_PIN_MUX_INVALID	An invalid pin mux value was supplied
STATUS_ERR_BAD_FORMAT	An invalid detection mode was requested

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 8-13. 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 8-14. 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 8-15. Parameters

Data direction	Parameter name	Description
[in]	channel	External Interrupt channel index to check.

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 8-16. 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 8-17. 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 8-18. Parameters

Data direction	Parameter name	Description
[in]	nmi_channel	External Interrupt NMI channel index to check.

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 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 8-19. 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 8-20. 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_NO_MEMORY	No free entries were found in the registration table.

Function extint_unregister_callback()

Unregisters an asynchronous callback function with the driver.

```
enum status_code extint_unregister_callback(
  const extint_callback_t callback,
  const enum extint_callback_type type)
```

Unregisters an asynchronous callback with the EXTINT driver, removing it from the internal callback registration table.

Table 8-21. Parameters

Data direction	Parameter name	Description
[in]	callback	Pointer to the callback function to unregister
[in]	type	Type of callback function to unregister

Returns Status of the de-registration operation.

Table 8-22. Return Values

Return value	Description
STATUS_OK	The callback was Unregistered successfully.

Return value	Description
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied.
STATUS_ERR_BAD_ADDRESS	No matching entry was found in the registration table.

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 uint32_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 8-23. 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 8-24. 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 uint32_t channel,
  const enum extint_callback_type type)
```

Disables asynchronous callbacks for a given logical external interrupt channel and type.

Table 8-25. 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 8-26. Return Values

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

8.6.5 Enumeration Definitions

Callback configuration and initialization

Enum extint_callback_type

Enum for the possible callback types for the EXTINT module.

Table 8-27. 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)

Enum extint_detect

Enum for the possible signal edge detection modes of the External Interrupt Controller module.

Table 8-28. 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.

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 8-29. Members

Enum value	Description
EXTINT_PULL_UP	Internal pull-up resistor is enabled on the pin.

Enum value	Description
EXTINT_PULL_DOWN	Internal pull-down resistor is enabled on the pin.
EXTINT_PULL_NONE	Internal pull resistor is disconnected from the pin.

8.7 Extra Information for EXTINT Driver

8.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
EIC	External Interrupt Controller
MUX	Multiplexer
NMI	Non-Maskable Interrupt

8.7.2 Dependencies

This driver has the following dependencies:

- System Pin Multiplexer 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
Initial Release

8.8 Examples for EXTINT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 External Interrupt Driver (EXTINT). 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 EXTINT - Basic
- Quick Start Guide for EXTINT - Callback

8.8.1 Quick Start Guide for EXTINT - Basic

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.

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()`):

```
extint_enable();
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.

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

```
extint_chan_get_config_defaults(&config_extint_chan);
```

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);
```

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);
```

8.8.2 Quick Start Guide for EXTINT - Callback

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.

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,
            EXTINT_CALLBACK_TYPE_DETECT);
    extint_chan_enable_callback(BUTTON_0_EIC_LINE,
            EXTINT_CALLBACK_TYPE_DETECT);
}

void extint_detection_callback(
        uint32_t channel)
{
    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()`):

```
extint_enable();
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.

Note

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

```
extint_chan_get_config_defaults(&config_extint_chan);
```

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,
        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(
        uint32_t channel)
{
    bool pin_state = port_pin_get_input_level(BUTTON_0_PIN);
    port_pin_set_output_level(LED_0_PIN, pin_state);
}
```

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.

9. SAM D20 I2C Bus Driver (SERCOM I2C)

This driver for SAM D20 devices provides an interface for the configuration and management of the device's SERCOM I^{2}C module, for the transfer of data via an I^{2}C bus. The following driver API modes are covered by this manual:

- Master Mode Polled APIs
- Master Mode Callback APIs
- Slave Mode Polled APIs
- Slave Mode Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

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.

9.2 Module Overview

The outline of this section is as follows:

- Functional Description
- Bus Topology
- Transactions
- Multi Master
- Bus States
- Bus Timing
- Operation in Sleep Modes

9.2.1 Functional Description

The I^{2}C provides a simple two-wire bidirectional bus consisting of a wired-AND type serial clock line (SCA) and a wired-AND type serial data line (SDA).

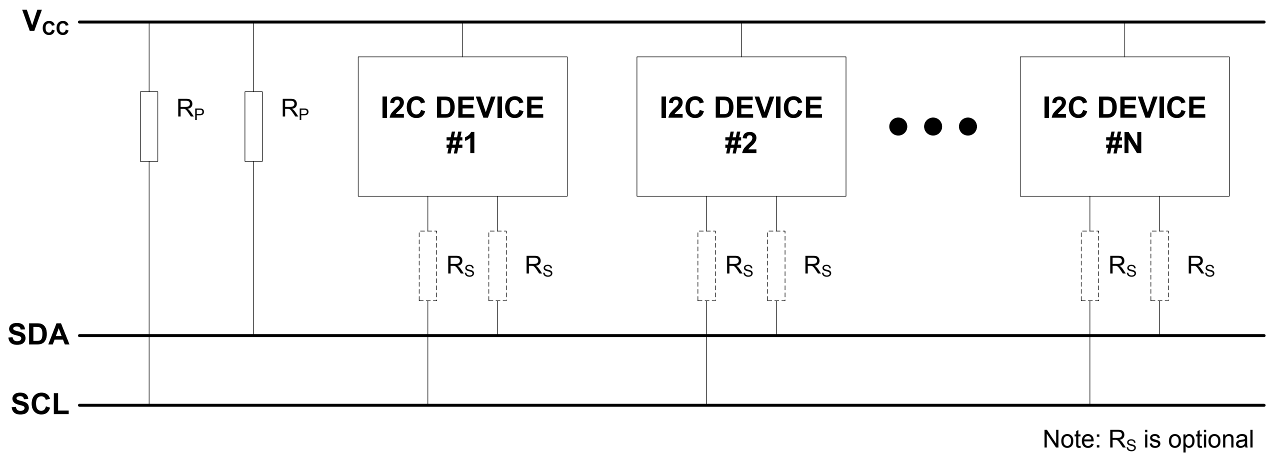
The I^{2}C bus provides a simple, but efficient method of interconnecting multiple master and slave devices. An arbitration mechanism is provided for resolving bus ownership between masters, as only one master device may own the bus at any given time. The arbitration mechanism relies on the wired-AND 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.

9.2.2 Bus Topology

The I^{2}C bus topology is illustrated in Figure 9-1: I2C bus topology. The pull-up resistors (Rs) will provide a high level on the bus lines when none of the I^{2}C devices are driving the bus. These are optional, and can be replaced with a constant current source.

Figure 9-1. I2C bus topology



9.2.3 Transactions

The I^{2}C 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.

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.

Data Packets

Data packets are nine bits long, consisting of one 8-bit data byte, and an acknowledgment bit. Data packets follow either an address packet or another data packet on the bus.

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 9-2: I2C Packet Read. 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 9-2. 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 9-3: I2C Packet Write. 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 9-3. 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

Packet Timeout

When a master sends an I^{2}C 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.

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.

9.2.4 Multi Master

In a multi master environment, arbitration of the bus is important, as only one master can own the bus at any point.

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.

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.

9.2.5 Bus States

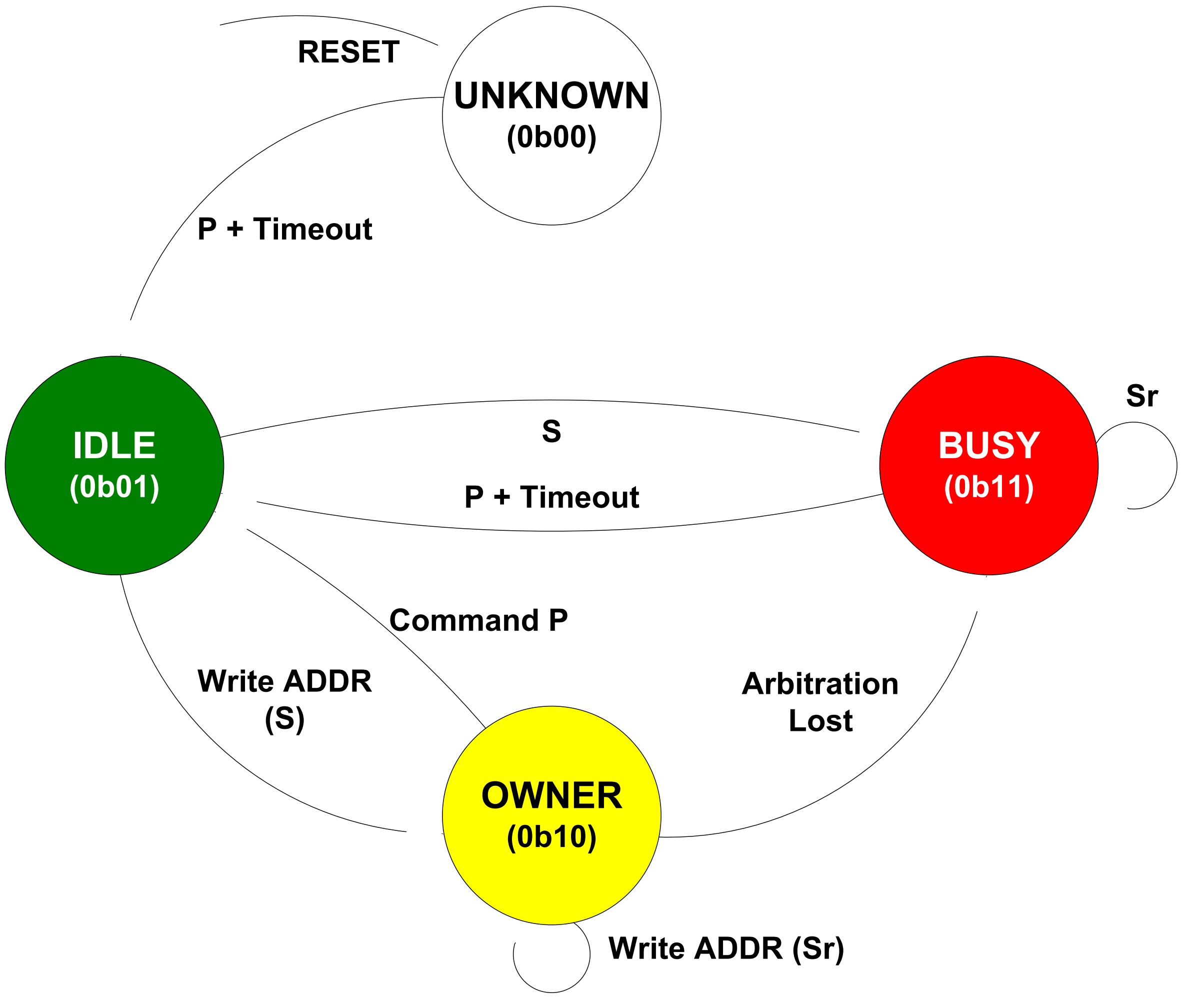
As the I^{2}C bus is limited to one transaction at the time, a master that wants to perform a bus transaction must wait until the bus is free. Because of this, it is necessary for all masters in a multi-master system to know the current status of the bus to be able to avoid conflicts and to ensure data 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 9-4: I2C bus state diagram.

- S: Start condition
- P: Stop condition
- Sr: Repeated start condition

Figure 9-4. I2C bus state diagram



9.2.6 Bus Timing

Inactive bus timeout for the master and SDA hold time is configurable in the drivers.

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.

SDA Hold Timeout

When using the I^2C 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.

9.2.7 Operation in Sleep Modes

The I^2C 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 9-1: I2C standby operations.

Table 9-1. I2C standby operations

Run in standby	Slave	Master
false	**Disabled, all reception is dropped**	**GCLK disabled when master is idle**
true	**Wake on address match when enabled**	**GCLK enabled while in sleep modes**
false	Disabled, all reception is dropped	GCLK disabled when master is idle
true	Wake on address match when enabled	GCLK enabled while in sleep modes

9.3 Special Considerations

9.3.1 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.

9.4 Extra Information

For extra information see Extra Information for SERCOM I2C Driver.

9.5 Examples

For a list of examples related to this driver, see Examples for SERCOM I2C Driver.

9.6 API Overview

9.6.1 Structure Definitions

Struct i2c_master_config

This is the configuration structure for the I^2C 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 9-2. Members

Type	Name	Description
enum i2c_master_baud_rate	baud_rate	Baud rate for I2C operations
uint16_t	buffer_timeout	Timeout for packet write to wait for slave
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
enum i2c_master_start_hold_time	start_hold_time	Bus hold time after start signal on data line
uint16_t	unknown_bus_state_timeout	Unknown bus state timeout

Struct i2c_master_module

SERCOM I^{2}C 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.

Struct i2c_packet

Structure to be used when transferring I^{2}C packets. Used both for master and slave driver modes.

Table 9-3. Members

Type	Name	Description
uint8_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

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 9-4. Members

Type	Name	Description
uint8_t	address	Address or upper limit of address range
uint8_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

Type	Name	Description
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
enum i2c_slave_sda_hold_time	sda_hold_time	SDA hold time with respect to the negative edge of SCL

Struct i2c_slave_module

SERCOM I^{2}C 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.

9.6.2 Macro Definitions

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 (1UL << 0)
```

Address Match

Note Should only be cleared internally by driver.

Macro I2C_SLAVE_STATUS_DATA_READY

```
#define I2C_SLAVE_STATUS_DATA_READY (1UL << 1)
```

Data Ready

Macro I2C_SLAVE_STATUS_STOP_RECEIVED

```
#define I2C_SLAVE_STATUS_STOP_RECEIVED (1UL << 2)
```

Stop Received

Macro I2C_SLAVE_STATUS_CLOCK_HOLD

```
#define I2C_SLAVE_STATUS_CLOCK_HOLD (1UL << 3)
```

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 (1UL << 4)
```

SCL Low Timeout

Macro I2C_SLAVE_STATUS_REPEATED_START

```
#define I2C_SLAVE_STATUS_REPEATED_START (1UL << 5)
```

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 (1UL << 6)
```

Received not acknowledge

Note Cannot be cleared

Macro I2C_SLAVE_STATUS_COLLISION

```
#define I2C_SLAVE_STATUS_COLLISION (1UL << 7)
```

Transmit Collision

Macro I2C_SLAVE_STATUS_BUS_ERROR

```
#define I2C_SLAVE_STATUS_BUS_ERROR (1UL << 8)
```

Bus error

9.6.3 Function Definitions

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 9-5. Parameters

Data direction	Parameter name	Description
[inout]	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 9-6. Parameters

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

Data direction	Parameter name	Description
[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 9-7. Parameters

Data direction	Parameter name	Description
[inout]	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 9-8. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to the software module struct
[in]	callback_type	Callback type to disable

Read and Write, Interrupt-Driven

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_packet *const packet)
```

Reads a data packet from the specified slave address on the I^{2}C bus. This is the non-blocking equivalent of i2c_master_read_packet_wait.

Table 9-9. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of starting reading I^{2}C packet.

Table 9-10. 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_packet *const packet)
```

Reads a data packet from the specified slave address on the I^{2}C 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 9-11. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of starting reading I^{2}C packet.

Table 9-12. 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_packet_job()

Initiates a write packet operation.

```
enum status_code i2c_master_write_packet_job(
  struct i2c_master_module *const module,
  struct i2c_packet *const packet)
```

Writes a data packet to the specified slave address on the I^{2}C bus. This is the non-blocking equivalent of i2c_master_write_packet_wait.

Table 9-13. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of starting writing I^{2}C packet job.

Table 9-14. 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_packet *const packet)
```

Writes a data packet to the specified slave address on the I^{2}C 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 9-15. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of starting writing I^{2}C packet job.

Table 9-16. 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 9-17. Parameters

Data direction	Parameter name	Description
[inout]	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 9-18. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure

Returns Last status code from transfer operation.

Table 9-19. Return Values

Return value	Description
STATUS_OK	No error has occurred
STATUS_BUSY	If transfer is in progress
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

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 9-20. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to software module structure

Returns Status of the synchronization.

Table 9-21. 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

Table 9-22. 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 9-23. 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 9-24. 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 9-25. 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 9-26. 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 9-27. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module structure

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_packet *const packet)
```

Writes a packet to the master. This will wait for the master to issue a request.

Table 9-28. 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 9-29. Return Values

Return value	Description
STATUS_OK	Packet was written successfully
STATUS_ERR_DENIED	Start condition not received, another interrupt flag is set
STATUS_ERR_IO	There was an error in the previous transfer

Return value	Description
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_packet *const packet)
```

Reads a packet from the master. This will wait for the master to issue a request.

Table 9-30. 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 9-31. 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)
```

Waits for the master to issue a start condition on the bus. Note that this function does not check for errors in the last transfer, this will be discovered when reading or writing.

Table 9-32. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure

Returns Direction of the current transfer, when in slave mode.

Table 9-33. 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

Function i2c_slave_get_direction()

```
enum i2c_slave_direction i2c_slave_get_direction(
  struct i2c_slave_module *const module)
```

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 9-34. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the I2C slave software device struct

Returns Bitmask of status flags

Table 9-35. Return Values

Return value	Description
I2C_SLAVE_STATUS_ADDRESS_MATCH	A valid address has been received
I2C_SLAVE_STATUS_DATA_READY	A I2C 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 occured
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 I2C 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 9-36. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the I2C software device struct
[in]	status_flags	Bit mask of status flags to clear

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 9-37. Parameters

Data direction	Parameter name	Description
[inout]	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 9-38. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module structure

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 9-39. Parameters

Data direction	Parameter name	Description
[inout]	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 9-40. Parameters

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

Data direction	Parameter name	Description
[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 9-41. Parameters

Data direction	Parameter name	Description
[inout]	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 9-42. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to the software module struct
[in]	callback_type	Callback type to disable

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_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 [189] callback can be used to call this function.

Table 9-43. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of starting asynchronously reading I^{2}C packet.

Table 9-44. 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_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 [189] callback can be used to call this function.

Table 9-45. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of starting writing I^{2}C packet.

Table 9-46. 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 9-47. Parameters

Data direction	Parameter name	Description
[inout]	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 9-48. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module structure

Returns Status of job.

Table 9-49. 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

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 9-50. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to software module structure

Returns Status of the synchronization.

Table 9-51. 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

Table 9-52. 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 I^{2}C master device requested and sets the provided software module struct. Run this function before any further use of the driver.

Table 9-53. Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to software module struct

Data direction	Parameter name	Description
[in]	hw	Pointer to the hardware instance
[in]	config	Pointer to the configuration struct

Returns Status of initialization.

Table 9-54. 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 I^{2}C module and set the bus state to IDLE after the specified timeout period if no stop bit is detected.

Table 9-55. 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 I^{2}C module.

Table 9-56. 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 9-57. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module structure

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_packet *const packet)
```

Reads a data packet from the specified slave address on the I^{2}C 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 9-58. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of reading packet.

Table 9-59. 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_packet *const packet)
```

Reads a data packet from the specified slave address on the I^{2}C 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 9-60. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of reading packet.

Table 9-61. 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_write_packet_wait()

Writes data packet to slave.

```
enum status_code i2c_master_write_packet_wait(
  struct i2c_master_module *const module,
  struct i2c_packet *const packet)
```

Writes a data packet to the specified slave address on the I^{2}C 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 9-62. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of reading packet.

Table 9-63. Return Values

Return value	Description
STATUS_OK	If packet was read
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_packet *const packet)
```

Writes a data packet to the specified slave address on the I^{2}C 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 9-64. Parameters

Data direction	Parameter name	Description
[inout]	module	Pointer to software module struct
[inout]	packet	Pointer to I^{2}C packet to transfer

Returns Status of reading packet.

Table 9-65. Return Values

Return value	Description
STATUS_OK	If packet was read
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

Return value	Description
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 9-66. Parameters

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

9.6.4 Enumeration Definitions

Enum i2c_master_baud_rate

Values for standard I^{2}C speeds supported by the module. The driver will also support setting any value between 10 and 100kHz, 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.

Note Max speed is given by GCLK-frequency divided by 10, and lowest is given by GCLK-frequency divided by 510.

Table 9-67. Members

Enum value	Description
I2C_MASTER_BAUD_RATE_100KHZ	Baud rate at 100kHz
I2C_MASTER_BAUD_RATE_400KHZ	Baud rate at 400kHz

Enum i2c_master_callback

The available callback types for the I^{2}C master module.

Table 9-68. Members

Enum value	Description
I2C_MASTER_CALLBACK_WRITE_COMPLETE	Callback for packet write complete
I2C_MASTER_CALLBACK_READ_COMPLETE	Callback for packet read complete

Enum value	Description
I2C_MASTER_CALLBACK_ERROR	Callback for error

Enum i2c_master_interrupt_flag

Flags used when reading or setting interrupt flags.

Table 9-69. Members

Enum value	Description
I2C_MASTER_INTERRUPT_WRITE	Interrupt flag used for write
I2C_MASTER_INTERRUPT_READ	Interrupt flag used for read

Enum i2c_master_start_hold_time

Values for the possible I^{2}C master mode SDA internal hold times after start bit has been sent.

Table 9-70. 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

Enum i2c_slave_address_mode

Enum for the possible address modes.

Table 9-71. 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

Enum i2c_slave_callback

The available callback types for the I2C slave.

Table 9-72. 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

Enum value	Description
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

Enum i2c_slave_direction

Enum for the direction of a request.

Table 9-73. Members

Enum value	Description
I2C_SLAVE_DIRECTION_READ	Read
I2C_SLAVE_DIRECTION_WRITE	Write
I2C_SLAVE_DIRECTION_NONE	No direction

Enum i2c_slave_sda_hold_time

Enum for the possible SDA hold times with respect to the negative edge of SCL.

Table 9-74. 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

9.7 Extra Information for SERCOM I2C Driver

9.7.1 Acronyms

Table 9-75: Acronyms is a table listing the acronyms used in this module, along with their intended meanings.

Table 9-75. Acronyms

Acronym	Description
SDA	Serial Data Line
SCL	Serial Clock Line

9.7.2 Dependencies

The I^{2}C 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

Table 9-76: Module History 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 9-76: Module History.

Table 9-76. Module History

Changelog
Initial Release

9.8 Examples for SERCOM I2C Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 I2C Bus Driver (SERCOM I2C). 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 I2C Master module - Basic Use Case
- Quick Start Guide for the I2C Master module - Callback Use Case
- Quick Start Guide for the I2C Slave module - Basic Use Case
- Quick Start Guide for the I2C Slave module - Callback Use Case

9.8.1 Quick Start Guide for SERCOM I2C Master - Basic

In this use case, the I^2C will 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

Prerequisites

The device must be connected to an I^2C slave.

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:

```
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, SERCOM2, &config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}
```

- Add to user application `main()`:

```
system_init();

/* Configure device and enable. */
configure_i2c_master();

/* Timeout counter. */
uint16_t timeout = 0;

/* Init i2c packet. */
struct i2c_packet packet = {
    .address     = SLAVE_ADDRESS,
    .data_length = DATA_LENGTH,
    .data        = buffer,
};
```

Workflow

1. Initialize system.

```
system_init();
```

2. 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, SERCOM2, &config_i2c_master);

    i2c_master_enable(&i2c_master_instance);
}
```

   a. Create and initialize configuration structure.

```
struct i2c_master_config config_i2c_master;
i2c_master_get_config_defaults(&config_i2c_master);
```

b. Change settings in the configuration.

```
config_i2c_master.buffer_timeout = 10000;
```

c. Initialize the module with the set configurations.

```
i2c_master_init(&i2c_master_instance, SERCOM2, &config_i2c_master);
```

d. Enable the module.

```
i2c_master_enable(&i2c_master_instance);
```

3. Create a variable to see when we should stop trying to send packet.

```
uint16_t timeout = 0;
```

4. Create a packet to send:

```
struct i2c_packet packet = {
    .address     = SLAVE_ADDRESS,
    .data_length = DATA_LENGTH,
    .data        = buffer,
};
```

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;
    }
}
```

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.

9.8.2 Quick Start Guide for SERCOM I2C Master - Callback

In this use case, the I^{2}C will used and set up as follows:

- Master mode
- 100kHz operation speed
- Not operational in standby
- 65535 unknown bus state timeout value

Prerequisites

The device must be connected to an I^{2}C slave.

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 buffer[DATA_LENGTH] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07
};

static uint8_t buffer_reversed[DATA_LENGTH] = {
    0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00
};
```

Address of slave:

```
#define SLAVE_ADDRESS 0x12
```

Globally accessible module structure:

```
struct i2c_master_module i2c_master_instance;
```

Globally accessible packet:

```
struct i2c_packet 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;

    /* Initialize and enable device with config */
    while(i2c_master_init(&i2c_master_instance, SERCOM2, &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)
{
    /* Send every other packet with reversed data */
    if (packet.data[0] == 0x00) {
        packet.data = &buffer_reversed[0];
    } else {
        packet.data = &buffer[0];
    }

    /* Initiate new packet write */
    i2c_master_read_packet_job(module, &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():

```
system_init();

/* Configure device and enable. */
configure_i2c();
/* Configure callbacks and enable. */
configure_i2c_callbacks();
```

Workflow

1. Initialize system.

```
system_init();
```

2. Configure and enable module.

```
struct i2c_master_config config_i2c_master;
i2c_master_get_config_defaults(&config_i2c_master);
```

   a. Create and initialize configuration structure.

   ```
   struct i2c_master_config config_i2c_master;
   i2c_master_get_config_defaults(&config_i2c_master);
   ```

   b. Change settings in the configuration.

   ```
   config_i2c_master.buffer_timeout = 65535;
   ```

   c. Initialize the module with the set configurations.

   ```
   while(i2c_master_init(&i2c_master_instance, SERCOM2, &config_i2c_master) != STATUS_OK);
   ```

   d. Enable the module.

   ```
   i2c_master_enable(&i2c_master_instance);
   ```

3. Configure callback funtionality.

```
configure_i2c_callbacks();
```

   a. Register write complete callback.

   ```
   i2c_master_register_callback(&i2c_master_instance, i2c_write_complete_callback,
           I2C_MASTER_CALLBACK_WRITE_COMPLETE);
   ```

   b. Enable write complete callback.

   ```
   i2c_master_enable_callback(&i2c_master_instance,
           I2C_MASTER_CALLBACK_WRITE_COMPLETE);
   ```

4. Create a packet to send to slave.

```
packet.address     = SLAVE_ADDRESS;
packet.data_length = DATA_LENGTH;
packet.data        = buffer;
```

Implementation

Code

Add to user application main:

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

Workflow

1. Write packet to slave.

```
i2c_master_write_packet_job(&i2c_master_instance, &packet);
```

2. Infinite while loop, while waiting for interaction with slave.

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

Callback

Each time a packet is sent, the callback function will be called.

Workflow

- Write complete callback:

  1. Send every other packet in reversed orded.

  ```
  if (packet.data[0] == 0x00) {
      packet.data = &buffer_reversed[0];
  } else {
      packet.data = &buffer[0];
  }
  ```

  2. Write new packet to slave.

  ```
  i2c_master_read_packet_job(module, &packet);
  ```

9.8.3 Quick Start Guide for SERCOM I2C Slave - Basic

In this use case, the I^{2}C will used and set up as follows:

- Slave mode
- 100kHz operation speed
- Not operational in standby
- 10000 packet timeout value

Prerequisites

The device must be connected to an I^{2}C master.

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;
    config_i2c_slave.buffer_timeout = 1000;

    /* Initialize and enable device with config_i2c_slave */
    i2c_slave_init(&i2c_slave_instance, SERCOM1, &config_i2c_slave);

    i2c_slave_enable(&i2c_slave_instance);
}
```

Add to user application main():

```
system_init();

configure_i2c_slave();

enum i2c_slave_direction dir;
struct i2c_packet packet = {
    .address     = SLAVE_ADDRESS,
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
};
```

Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Configure and enable module:

   ```
   configure_i2c_slave();
   ```

   a. Create and initialize configuration structure.

      ```
      struct i2c_slave_config config_i2c_slave;
      i2c_slave_get_config_defaults(&config_i2c_slave);
      ```

b. 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;
```

c. Initialize the module with the set configurations.

```
i2c_slave_init(&i2c_slave_instance, SERCOM1, &config_i2c_slave);
```

d. Enable the module.

```
i2c_slave_enable(&i2c_slave_instance);
```

3. Create variable to hold transfer direction

```
enum i2c_slave_direction dir;
```

4. Create packet variable to transfer

```
struct i2c_packet packet = {
    .address     = SLAVE_ADDRESS,
    .data_length = DATA_LENGTH,
    .data        = write_buffer,
};
```

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);
}
```

9.8.4 Quick Start Guide for SERCOM I2C Slave - Callback

In this use case, the I^{2}C will used and set up as follows:

- Slave mode
- 100kHz operation speed
- Not operational in standby
- 10000 packet timeout value

Prerequisites

The device must be connected to an I^{2}C master.

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_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;
    /* Initialize and enable device with config. */
    i2c_slave_init(&i2c_slave_instance, SERCOM2, &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():

```
system_init();

/* Configure device and enable. */
configure_i2c_slave();
configure_i2c_slave_callbacks();
```

Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Configure and enable module:

   ```
   configure_i2c_slave();
   ```

   a. Create and initialize configuration structure.

      ```
      struct i2c_slave_config config_i2c_slave;
      i2c_slave_get_config_defaults(&config_i2c_slave);
      ```

   b. 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;
      ```

   c. Initialize the module with the set configurations.

      ```
      i2c_slave_init(&i2c_slave_instance, SERCOM2, &config_i2c_slave);
      ```

   d. Enable the module.

      ```
      i2c_slave_enable(&i2c_slave_instance);
      ```

3. Register and enable callback functions.

   ```
   configure_i2c_slave_callbacks();
   ```

   a. 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);
      ```

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 */
   }
   ```

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) {
     }
     ```

10. SAM D20 Non-Volatile Memory Driver (NVM)

This driver for SAM D20 devices 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 peripherals are used by this module:

- NVM (Non-Volatile Memory)

The outline of this documentation is as follows:

10.1 Prerequisites

There are no prerequisites for this module.

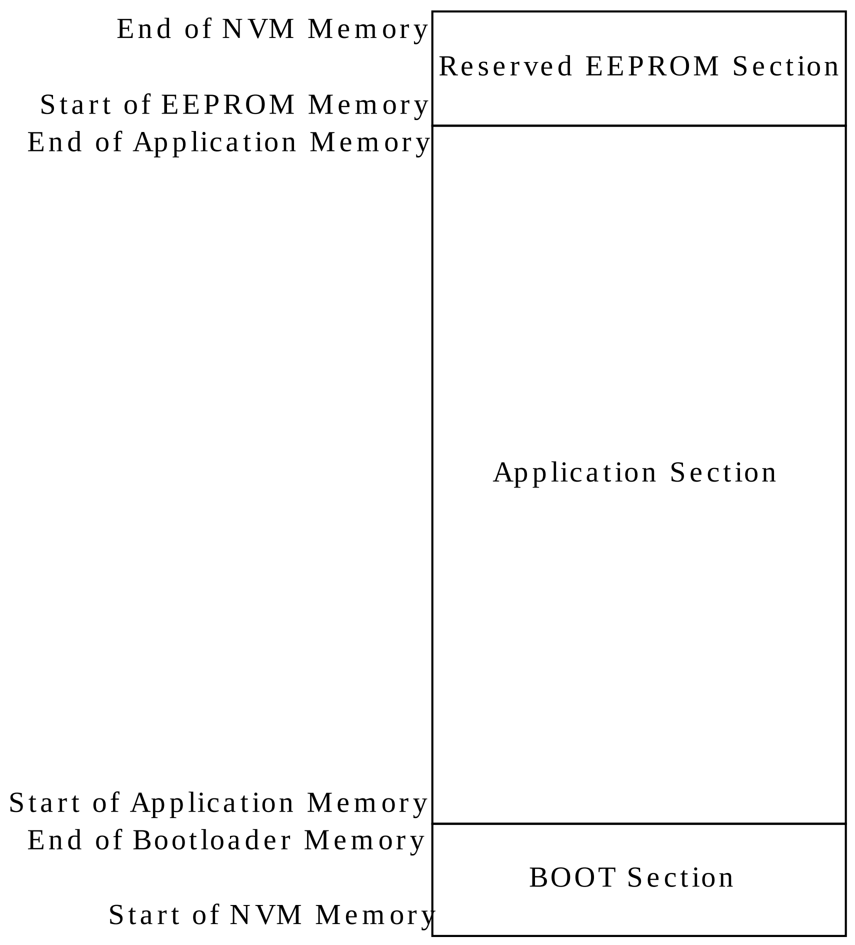
10.2 Module Overview

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.

10.2.1 Memory Regions

The NVM memory space of the SAM D20 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 10-1: Memory Regions.

Figure 10-1. 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.

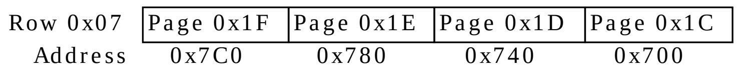
A NVM page number and address can be computed via the following equations:

$$PageNum = (RowNum \times 4) + PagePosInRow \tag{10-1}$$

$$PageAddr = PageNum \times PageSize \tag{10-2}$$

Figure 10-2: Memory Regions shows an example of the memory page and address values associated with logical row 7 of the NVM memory space.

Figure 10-2. Memory Regions



10.2.2 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 [214] 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 [214] or NVM_COMMAND_UNLOCK_REGION [214] 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 to be written, however the adjusted configuration will not take effect until the next device reset.

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.

10.2.3 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, or a buffer of data less than a full page is passed to the buffer write function.

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.

10.3 Special Considerations

10.3.1 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.

10.3.2 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.

10.3.3 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 [214] 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.

10.4 Extra Information for NVM

For extra information see Extra Information for NVM Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

10.5 Examples

For a list of examples related to this driver, see Examples for NVM Driver.

10.6 API Overview

10.6.1 Structure Definitions

Struct nvm_config

Configuration structure for the NVM controller within the device.

Table 10-1. Members

Type	Name	Description
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.[1]
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.

Notes: [1]If a partial page is to be written, a manual write command must be executed in either mode.

Struct nvm_parameters

Structure containing the memory layout parameters of the NVM module.

Table 10-2. 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	eeprom_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.

10.6.2 Function Definitions

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 until first NVM access
- Automatic page commit when full pages are written to
- Number of FLASH wait states left unchanged

Table 10-3. 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 a NVM controller configuration to the hardware module, and initializes the internal device struct

Table 10-4. 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 10-5. 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 10-6. Return Values

Return value	Description
true	If the hardware module is ready for a new command

Return value	Description
false	If the hardware module is busy executing a command

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 10-7. 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 10-8. 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.

Returns Status of the attempt to write a page.

Table 10-9. Return Values

Return value	Description
STATUS_OK	Requested NVM memory page was successfully read

Return value	Description
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 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 10-10. 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 10-11. 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.

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 10-12. 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 10-13. 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 10-14. Parameters

Data direction	Parameter name	Description
[in]	row_address	Address of the row to erase

Returns Status of the NVM row erase attempt.

Table 10-15. Return Values

Return value	Description
STATUS_OK	Requested NVM memory row was successfully erased

Return value	Description
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

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 a NVM page read or write operation.

Note The function will return before the execution of the given command is completed.

Table 10-16. 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 10-17. 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_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 10-18. Parameters

Data direction	Parameter name	Description
[in]	page_number	Page number to be checked

Returns Page lock status

Table 10-19. 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 10-20. 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

10.6.3 Enumeration Definitions

Enum nvm_command

Table 10-21. Members

Enum value	Description
NVM_COMMAND_ERASE_ROW	Erases the addressed memory row.

Enum value	Description
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. When in low power mode, all commands other than NVM_COMMAND_EXIT_LOW_POWER_MODE [214] will fail.
NVM_COMMAND_EXIT_LOW_POWER_MODE	Exit power reduction mode in the NVM controller to allow other NVM commands to be issued.

Enum nvm_error

Possible NVM controller error codes, which can be returned by the NVM controller after a command is issued.

Table 10-22. 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.

Enum nvm_sleep_power_mode

Power reduction modes of the NVM controller, to conserve power while the device is in sleep.

Table 10-23. Members

Enum value	Description
NVM_SLEEP_POWER_MODE_WAKEONACCESS	NVM controller exits low power mode on first access after sleep.

Enum value	Description
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.

10.7 Extra Information for NVM Driver

10.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
NVM	Non-Volatile Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory

10.7.2 Dependencies

This driver has the following dependencies:

- None

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 NVM Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Non-Volatile Memory Driver (NVM). 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 NVM - Basic

10.8.1 Quick Start Guide for NVM - Basic

In this use case, the NVM module is configured for:

- Power reduction mode enabled after sleep 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 the read it back into the same buffer.

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);

    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.

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

   ```
   nvm_get_config_defaults(&config_nvm);
   ```

3. Configure NVM controller with the created configuration struct settings.

   ```
   nvm_set_config(&config_nvm);
   ```

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.

Note This must be performed before writing new data into a NVM page.

   ```
   do
   {
       error_code = nvm_erase_row(
               100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE);
   } while (error_code == STATUS_BUSY);
   ```

5. Write the buffer of data to the previously erased page of the NVM.

Note The new data will be written to NVM memory automatically, as the NVM controller is configured in automatic page write mode.

```
do
{
    error_code = nvm_write_buffer(
            100 * NVMCTRL_ROW_PAGES * NVMCTRL_PAGE_SIZE,
            page_buffer, NVMCTRL_PAGE_SIZE);
} while (error_code == STATUS_BUSY);
```

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);
```

11. SAM D20 Peripheral Access Controller Driver (PAC)

This driver for SAM D20 devices 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 outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for PAC
- Examples
- API Overview

11.1 Prerequisites

There are no prerequisites for this module.

11.2 Module Overview

The SAM D20 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 Code Run-away 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.

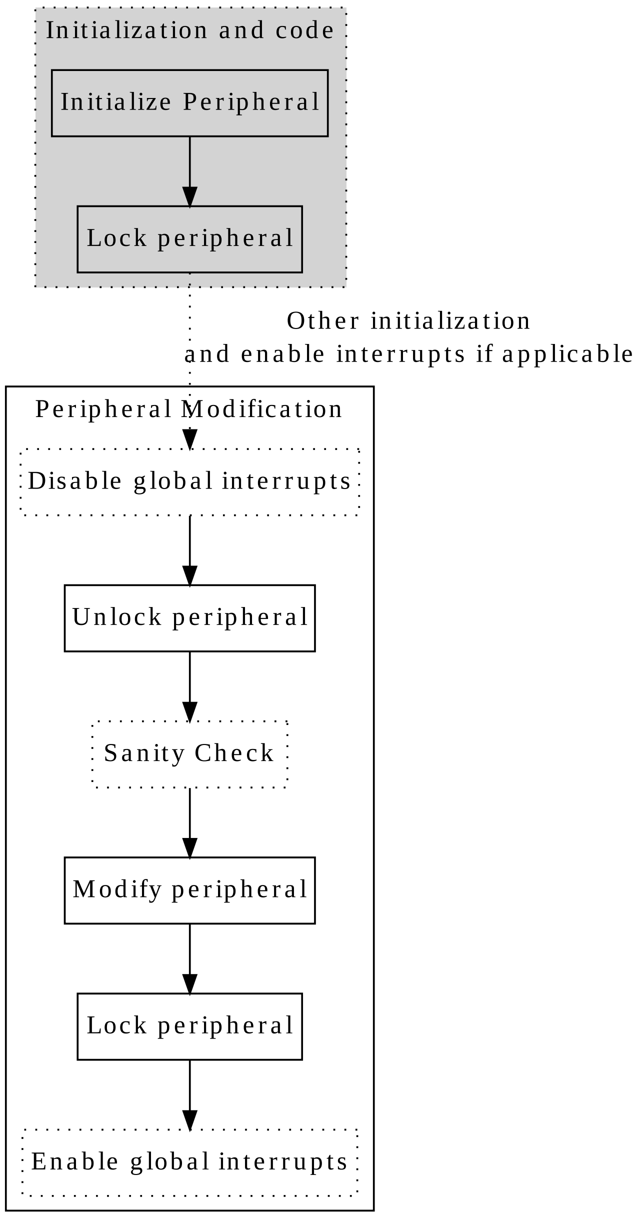
11.2.1 Locking Scheme

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 and non-maskable interrupt (NMI). 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 code run-away 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.

11.2.2 Recommended Implementation

A recommended implementation of the PAC can be seen in Figure 11-1: Recommended Implementation.

Figure 11-1. Recommended Implementation



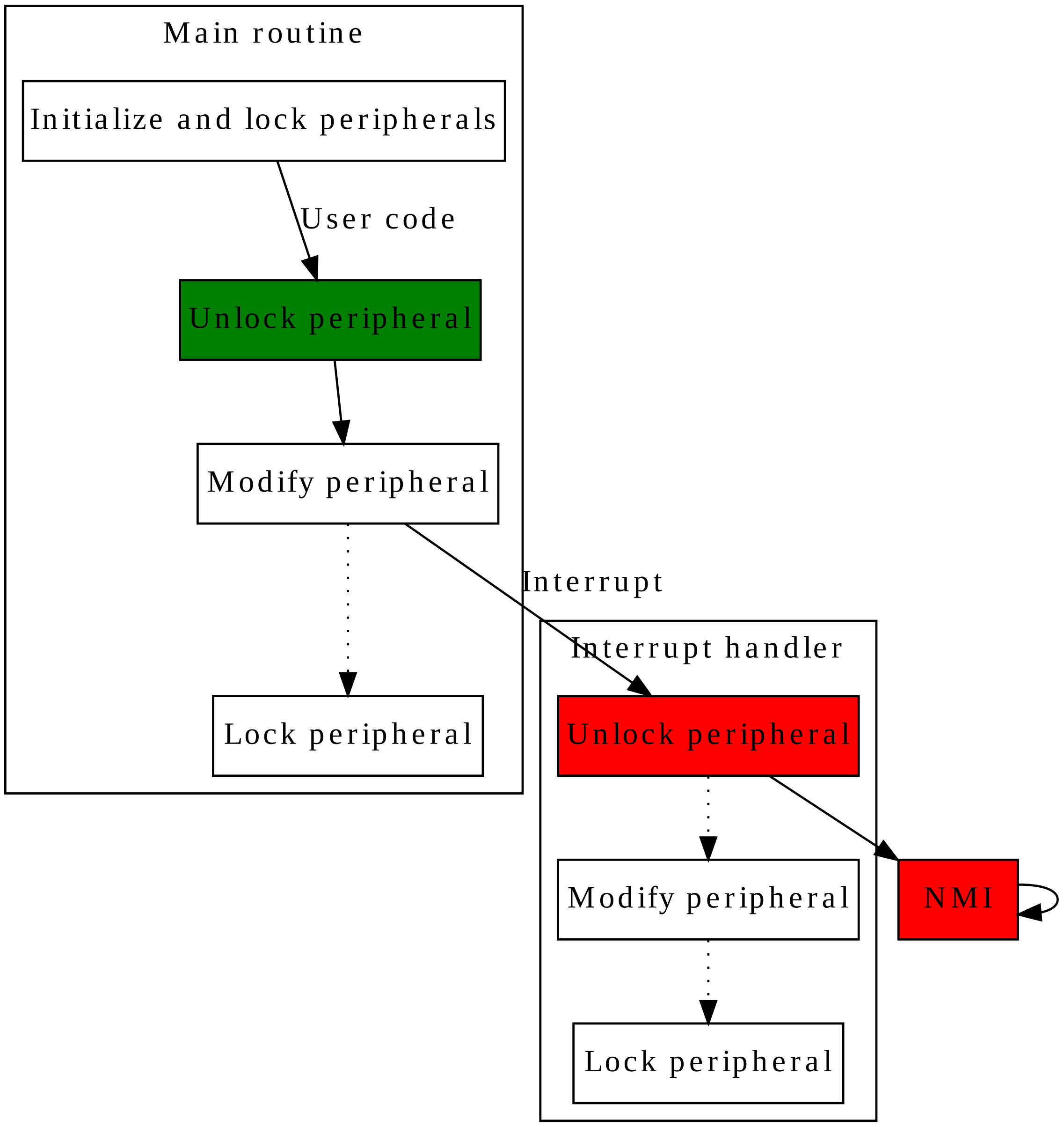
11.2.3 Why Disable Interrupts

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 11-2: Why Disable Interrupts.

Figure 11-2. Why Disable Interrupts

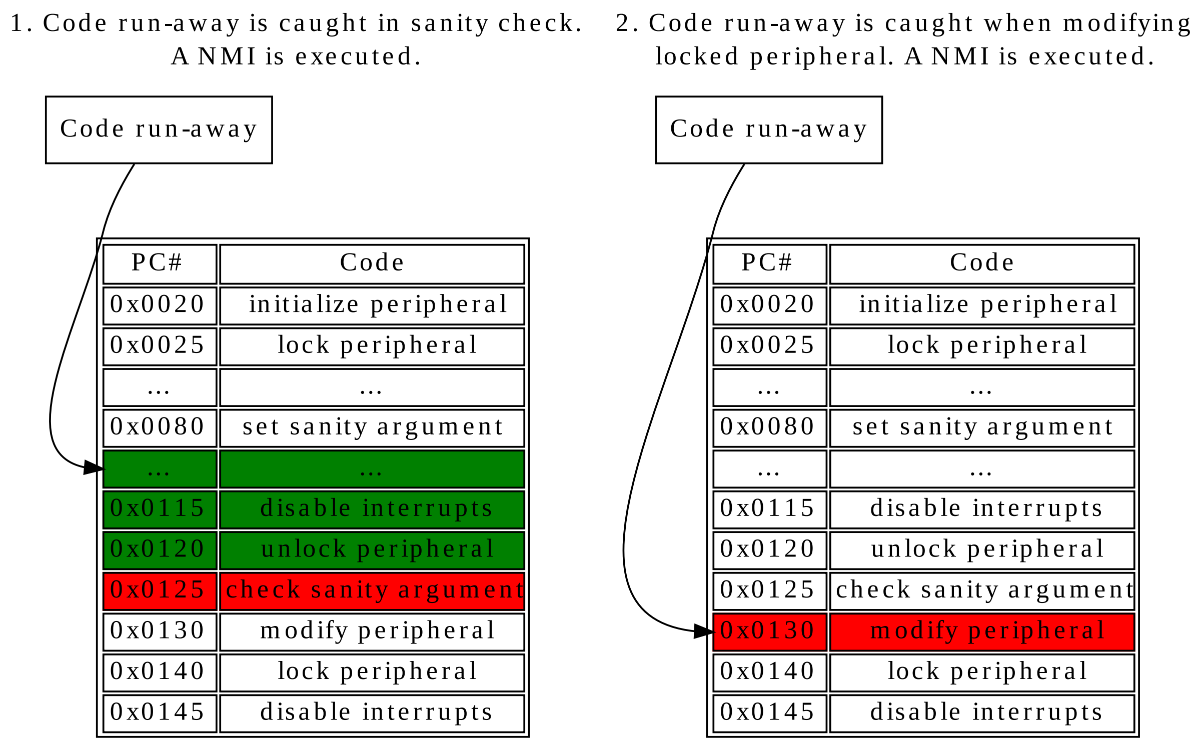


11.2.4 Code Run-away

Code run-away can be caused by the MCU being operated outside its specification, faulty code or EMI issues. If a code run-away occurs, it is favorable to catch the issue as soon as possible. With a correct implementation of the PAC, the code run-away can potentially be stopped.

A graphical example showing how a PAC implementation will behave for different circumstances of code run-away in shown in Figure 11-3: Code Run-away and Figure 11-4: Code Run-away.

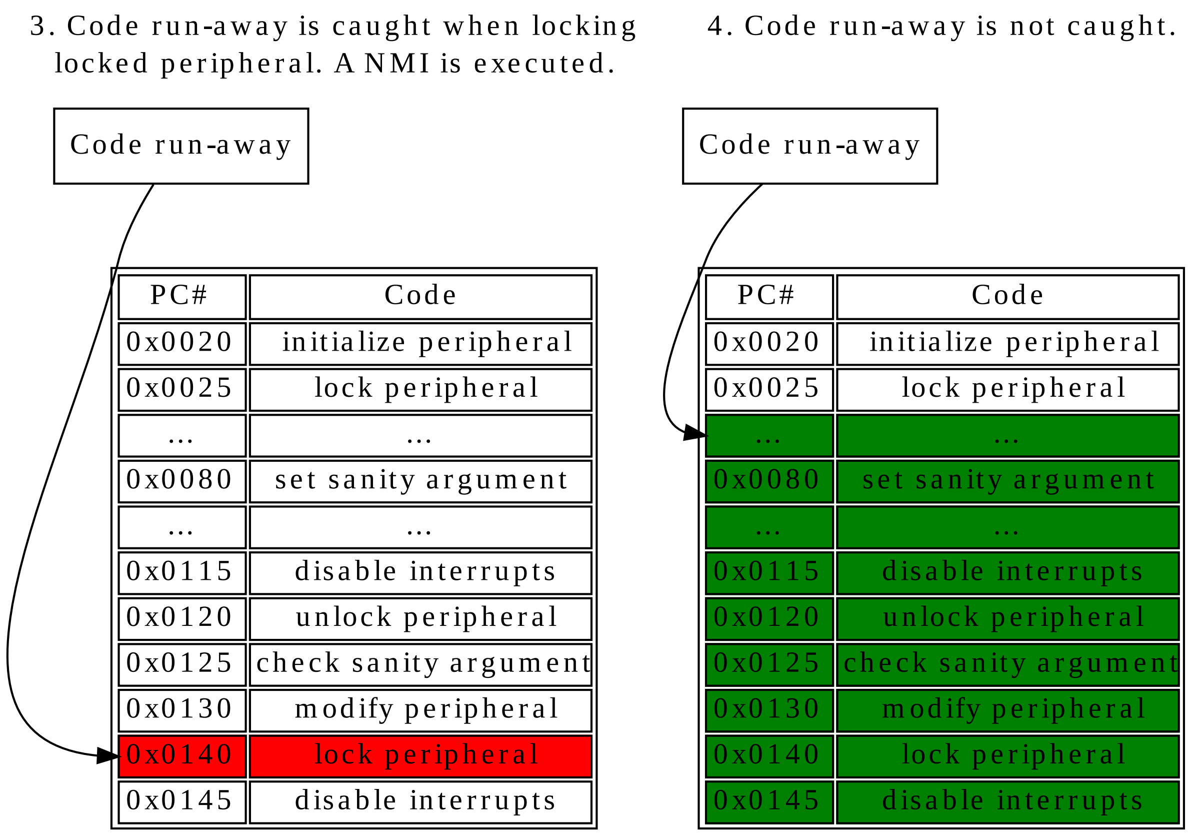
Figure 11-3. Code Run-away


1. Code run-away is caught in sanity check. A NMI is executed.
Code run-away
2. Code run-away is caught when modifying locked peripheral. A NMI is executed.
Code run-away

PC#	Code
0x0020	initialize peripheral
0x0025	lock peripheral
...	...
0x0080	set sanity argument
...	...
0x0115	disable interrupts
0x0120	unlock peripheral
0x0125	check sanity argument
0x0130	modify peripheral
0x0140	lock peripheral
0x0145	disable interrupts

PC#	Code
0x0020	initialize peripheral
0x0025	lock peripheral
...	...
0x0080	set sanity argument
...	...
0x0115	disable interrupts
0x0120	unlock peripheral
0x0125	check sanity argument
0x0130	modify peripheral
0x0140	lock peripheral
0x0145	disable interrupts

Figure 11-4. Code Run-away



In the example, green indicates that the command is allowed, red indicates where the code run-away will be caught, and the arrow where the code run-away enters the application. In special circumstances, like example 4 above, the code run-away will not be caught. However, the protection scheme will greatly enhance peripheral configuration security from being affected by code run-away.

Key-Argument

To protect the module functions against code run-away themselves, a key is required as one of the input arguments. The key-argument will make sure that code run-away 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.

11.2.5 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.

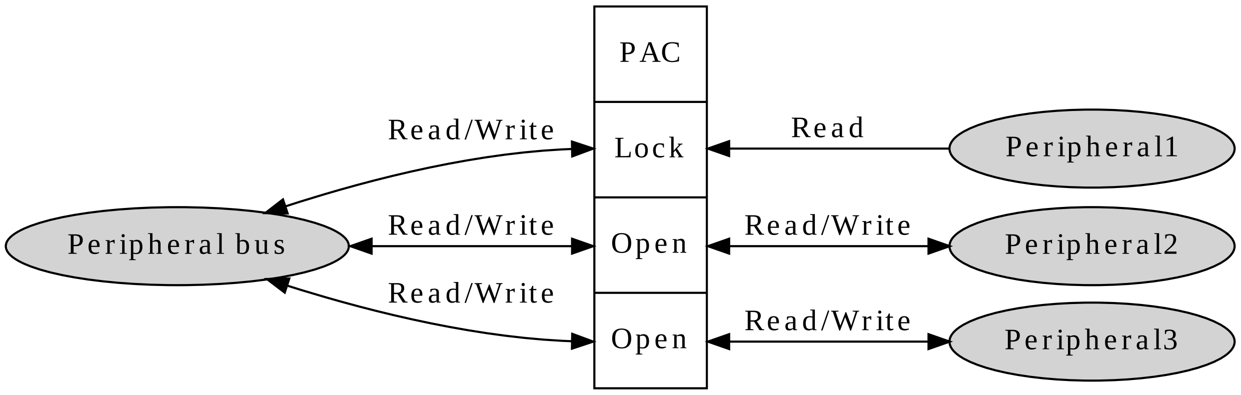
11.2.6 Use of __no_inline

All function for the given modules are specified to be `__no_inline`. This increases security as it decreases the probability that a return call is directed at the correct location.

11.2.7 Physical Connection

Figure 11-5: Physical Connection shows how this module is interconnected within the device.

Figure 11-5. Physical Connection



11.3 Special Considerations

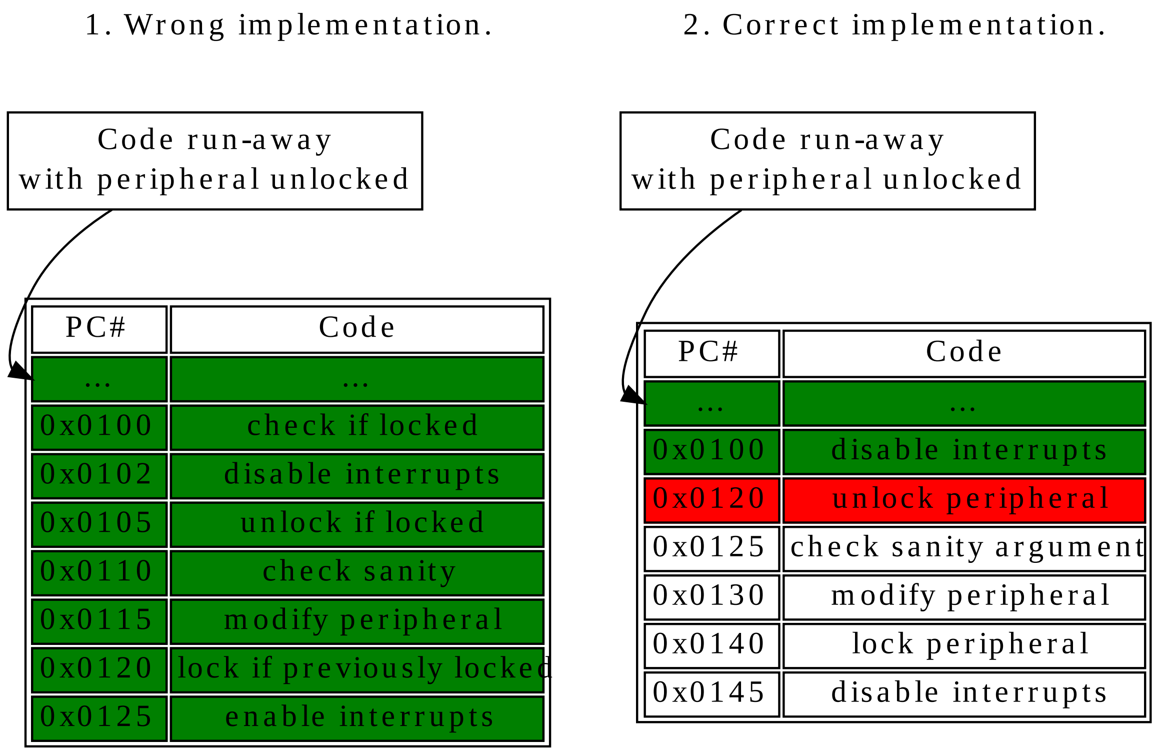
11.3.1 Non-Writable Registers

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.

11.3.2 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 11-6: Reading Lock State.

Figure 11-6. Reading Lock State


1. Wrong implementation.
2. Correct implementation.
Code run-away with peripheral unlocked
Code run-away with peripheral unlocked

PC#	Code
...	...
0x0100	check if locked
0x0102	disable interrupts
0x0105	unlock if locked
0x0110	check sanity
0x0115	modify peripheral
0x0120	lock if previously locked
0x0125	enable interrupts

PC#	Code
...	...
0x0100	disable interrupts
0x0120	unlock peripheral
0x0125	check sanity argument
0x0130	modify peripheral
0x0140	lock peripheral
0x0145	disable interrupts

In the left figure above, one can see the code run-away continues as all illegal operations are conditional. On the right side figure, the code run-away is caught as it tries to unlock the peripheral.

11.4 Extra Information for PAC

For extra information see Extra Information for PAC Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

11.5 Examples

For a list of examples related to this driver, see Examples for PAC Driver.

11.6 API Overview

11.6.1 Macro Definitions

Macro SYSTEM_PERIPHERAL_ID

```
#define SYSTEM_PERIPHERAL_ID(peripheral) \
  ID_##peripheral
```

Retrieves the ID of a specified peripheral name, giving its peripheral bus location.

Table 11-1. Parameters

Data direction	Parameter name	Description
[in]	peripheral	Name of the peripheral instance

Returns Bus ID of the specified peripheral instance.

11.6.2 Function Definitions

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 hard fault exception, and terminate program execution.

Table 11-2. 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 11-3. 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 hard fault exception, and terminate program execution.

Table 11-4. 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 11-5. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked.
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied.

11.7 List of Non-Write Protected Registers

Look in device datasheet peripheral's subsection "Register Access Protection" to see which is actually availeble 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	INTFALG
	STATUS
	DATA
TC	INTFLAG
	STATUS
TCE	INTFLAG
	STATUS
WDT	INTFLAG
	STATUS

Module	Non-write protected register
	(CLEAR)

11.8 Extra Information for PAC Driver

11.8.1 Acronyms

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

Acronym	Description
NMI	Non-Maskable Interrupt
PAC	Peripheral Access Controller
WDT	Watch Dog Timer

11.8.2 Dependencies

This driver has the following dependencies:

- None

11.8.3 Errata

There are no errata related to this driver.

11.8.4 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

11.9 Examples for PAC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Peripheral Access Controller Driver (PAC). 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 PAC - Basic

11.9.1 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.

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();
```

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));

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 Hard Fault exception.

```
system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
        ~SYSTEM_PERIPHERAL_ID(PORT));
```

3. Enable global interrupts.

```
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. SAM D20 Pin Multiplexer Driver (PINMUX)

This driver for SAM D20 devices 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 peripherals are used by this module:

- PORT (Port I/O Management)

Physically, the modules are interconnected within the device as shown in the following diagram:

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for pinmux
- Examples
- API Overview

12.1 Prerequisites

There are no prerequisites for this module.

12.2 Module Overview

The SAM D20 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,

12.2.1 Physical and Logical GPIO Pins

SAM D20 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.

12.2.2 Peripheral Multiplexing

SAM D20 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 I^2C module). When a pin is configured in GPIO mode, other peripherals connected to the same pin will be disabled.

12.2.3 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:

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.

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.

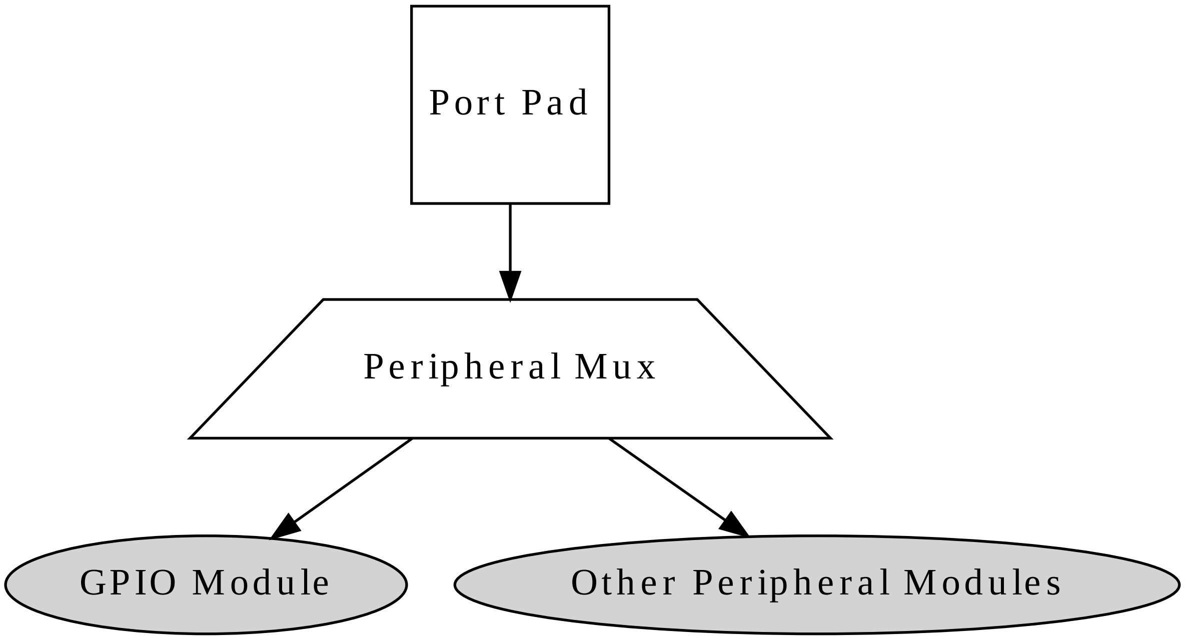
Input Sample Mode

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.

12.2.4 Physical Connection

Figure 12-1: Physical Connection shows how this module is interconnected within the device:

Figure 12-1. Physical Connection



12.3 Special Considerations

The SAM D20 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 four 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.

12.4 Extra Information for pinmux

For extra information see Extra Information for SYSTEM PINMUX Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

12.5 Examples

For a list of examples related to this driver, see Examples for SYSTEM PINMUX Driver.

12.6 API Overview

12.6.1 Structure Definitions

Struct system_pinmux_config

Configuration structure for a port pin instance. This structure should be structure should be initialized by the system_pinmux_get_config_defaults() function before being modified by the user application.

Table 12-1. 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.

12.6.2 Macro Definitions

Macro SYSTEM_PINMUX_GPIO

```
#define SYSTEM_PINMUX_GPIO (1 << 7)
```

Peripheral multiplexer index to select GPIO mode for a pin.

12.6.3 Function Definitions

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 12-2. 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 12-3. 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 12-4. 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.

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 12-5. 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 12-6. 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.

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 12-7. 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.

Function system_pinmux_group_set_output_slew_rate()

Configures the output slew rate mode for a group of pins.

```
void system_pinmux_group_set_output_slew_rate(
  PortGroup *const port,
  const uint32_t mask,
  const enum system_pinmux_pin_slew_rate mode)
```

Configures the output slew rate mode for a group of pins, to control the speed at which the physical output pin can react to logical changes of the I/O pin value.

Table 12-8. 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 slew rate mode to configure.

Function system_pinmux_group_set_output_drive()

Configures the output driver mode for a group of pins.

```
void system_pinmux_group_set_output_drive(
  PortGroup *const port,
  const uint32_t mask,
  const enum system_pinmux_pin_drive mode)
```

Configures the output driver mode for a group of pins, to control the pad behavior.

Table 12-9. 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 pad output driver mode to configure.

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 12-10. 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 12-11. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure.
[in]	mode	New pin sampling mode to configure.

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 12-12. 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.

Function system_pinmux_pin_set_output_slew_rate()

Configures the output slew rate mode for a GPIO pin.

```
void system_pinmux_pin_set_output_slew_rate(
  const uint8_t gpio_pin,
  const enum system_pinmux_pin_slew_rate mode)
```

Configures the output slew rate mode for a GPIO output, to control the speed at which the physical output pin can react to logical changes of the I/O pin value.

Table 12-13. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure.
[in]	mode	New pin slew rate mode to configure.

Function system_pinmux_pin_set_output_drive()

Configures the output driver mode for a GPIO pin.

```
void system_pinmux_pin_set_output_drive(
  const uint8_t gpio_pin,
  const enum system_pinmux_pin_drive mode)
```

Configures the output driver mode for a GPIO output, to control the pad behavior.

Table 12-14. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to configure.
[in]	mode	New pad output driver mode to configure.

12.6.4 Enumeration Definitions

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 12-15. 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_READB	The pin's output and input buffers should both be enabled, so that the pin state can be set and read back.

Enum system_pinmux_pin_drive

Enum for the possible output drive modes for the port pin configuration structure, to indicate the output mode the pin should use.

Table 12-16. Members

Enum value	Description
SYSTEM_PINMUX_PIN_DRIVE_TOTEM	Use totem pole output drive mode.

Enum value	Description
SYSTEM_PINMUX_PIN_DRIVE_OPEN_DRAIN	Use open drain output drive mode.

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 12-17. 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.

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 12-18. 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.

Enum system_pinmux_pin_slew_rate

Enum for the possible output drive slew rates for the port pin configuration structure, to indicate the driver slew rate the pin should use.

Table 12-19. Members

Enum value	Description
SYSTEM_PINMUX_PIN_SLEW_RATE_NORMAL	Normal pin output slew rate.
SYSTEM_PINMUX_PIN_SLEW_RATE_LIMITED	Enable slew rate limiter on the pin.

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 12-20. Members

Enum value	Description
SYSTEM_PINMUX_PIN_STRENGTH_NORMAL	Normal output driver strength.
SYSTEM_PINMUX_PIN_STRENGTH_HIGH	High current output driver strength.

12.7 Extra Information for SYSTEM PINMUX Driver

12.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
GPIO	General Purpose Input/Output
MUX	Multiplexer

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 SYSTEM PINMUX Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Pin Multiplexer Driver (PINMUX). 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 SYSTEM PINMUX - Basic

12.8.1 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.

Setup

Prerequisites

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

Use Case

Code

Copy-paste the following code to your user 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);

system_pinmux_pin_set_input_sample_mode(10,
        SYSTEM_PINMUX_PIN_SAMPLE_ONDEMAND);

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

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.

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

   ```
   system_pinmux_get_config_defaults(&config_pinmux);
   ```

3. Adjust the configuration struct to request an input pin with pullup 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);
   ```

5. 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);
   ```

13. SAM D20 Port Driver (PORT)

This driver for SAM D20 devices 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 peripherals are used by this module:

- PORT (GPIO Management)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for PORT
- 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.

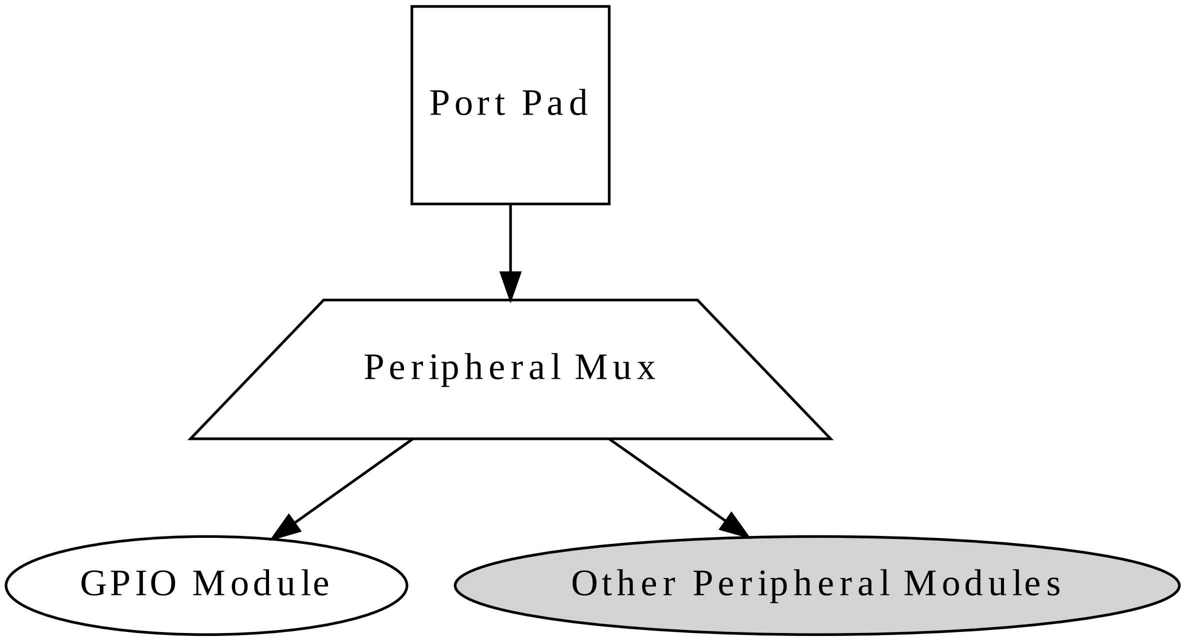
13.2.1 Physical and Logical GPIO Pins

SAM D20 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.

13.2.2 Physical Connection

Figure 13-1: Physical Connection shows how this module is interconnected within the device.

Figure 13-1. Physical Connection



13.3 Special Considerations

The SAM D20 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 PORT

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

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.

Type	Name	Description
enum port_pin_pull	input_pull	Port pull-up/pull-down for input pins.

13.6.2 Macro Definitions

PORT Alias Macros

Macro PORTA

```
#define PORTA PORT->Group[0]
```

Convenience definition for GPIO module group A on the device (if available).

Macro PORTB

```
#define PORTB PORT->Group[1]
```

Convenience definition for GPIO module group B on the device (if available).

Macro PORTC

```
#define PORTC PORT->Group[2]
```

Convenience definition for GPIO module group C on the device (if available).

Macro PORTD

```
#define PORTD PORT->Group[3]
```

Convenience definition for GPIO module group D on the device (if available).

13.6.3 Function Definitions

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 13-2. 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 13-3. 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 13-4. 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) 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 13-5. 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 13-6. 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 toggle.

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 pullup enabled

Table 13-7. 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 13-8. 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 13-9. 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.

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 13-10. 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 13-11. 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 13-12. 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 13-13. Parameters

Data direction	Parameter name	Description
[in]	gpio_pin	Index of the GPIO pin to toggle.

13.6.4 Enumeration Definitions

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 13-14. 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.

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 13-15. 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
Initial Release

13.8 Examples for PORT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Port Driver (PORT). 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 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.

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.

Note

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

```
port_get_config_defaults(&config_port_pin);
```

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 GPIO10 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.

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.

```
config_port_pin.direction = PORT_PIN_DIR_OUTPUT;
```

6. Configure GPIO11 with the initialized pin configuration struct, to enable the output driver on the pin.

```
port_pin_set_config(LED_0_PIN, &config_port_pin);
```

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 GPIO10, 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 GPIO11, 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 D20 RTC Count Driver (RTC COUNT)

This driver for SAM D20 devices 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 peripherals are used by this module:

- RTC (Real Time Clock)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for RTC COUNT
- Examples
- API Overview

14.1 Prerequisites

There are no prerequisites for this module.

14.2 Module Overview

The RTC module in the SAM D20 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 6 configurable compare values
- 32-bit counter mode
  - Clear counter value on match
  - Up to 4 configurable compare values

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:

$$COUNT_{MAX} = 2^{32} - 1 \tag{14-1}$$

for 32-bit counter mode, and

$$COUNT_{MAX} = 2^{16} - 1 \tag{14-2}$$

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 8 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:

$$f_{PERIODIC} = \frac{f_{ASY}}{2^{n+3}} \tag{14-3}$$

Where

$$f_{ASY} \tag{14-4}$$

refers to the *asynchronous* clock 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 1 KHz, the formula results in the values shown in Table 14-1: RTC event frequencies for each prescaler bit using a 1KHz clock.

Table 14-1. RTC event frequencies for each prescaler bit using a 1KHz clock

n	Periodic event
7	1 Hz
6	2 Hz
5	4 Hz
4	8 Hz
3	16 Hz
2	32 Hz
1	64 Hz
0	128 Hz

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 1 Hz 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 (max 127) over 976 of these periods. The corresponding correction in PPM will be given by:

$$Correction(PPM) = \frac{VALUE}{999424} 10^{6} \tag{14-5}$$

The RTC clock will tick faster if provided with a positive correction value, and slower when given a negative correction value.

14.4 Special Considerations

14.4.1 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 32 KHz RC-oscillator with a prescaler of 32, giving a resulting clock frequency of 1 KHz to the RTC. When the internal RTC prescaler is set to 1024, this yields an end-frequency of 1 Hz.

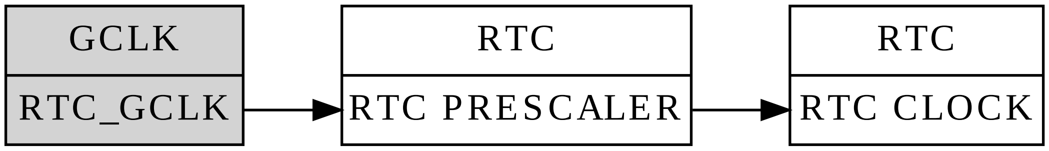
The implementer also has the option to set other end-frequencies. Table 14-2: RTC output frequencies from allowable input clocks lists the available RTC frequencies for each possible GCLK and RTC input prescaler options.

Table 14-2. RTC output frequencies from allowable input clocks

End-frequency	GCLK prescaler	RTC Prescaler
32 KHz	1	1
1 KHz	32	1
1 Hz	32	1024

The overall RTC module clocking scheme is shown in Figure 14-1: Clock Setup.

Figure 14-1. Clock Setup



14.5 Extra Information for RTC COUNT

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

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 14-3. 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.

Type	Name	Description
uint32_t	compare_values[]	Array of Compare values. Not all Compare values are available in 32-bit mode.
bool	continuously_update	Continuously update the counter value so no synchronization is needed for reading.
enum rtc_count_mode	mode	Select the operation mode of the RTC.
enum rtc_count_prescaler	prescaler	Input clock prescaler for the RTC module.

Struct rtc_count_events

Event flags for the rtc_count_enable_events() and rtc_count_disable_events().

Table 14-4. 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 (see Periodic Events).

14.7.2 Function Definitions

Configuration and initialization

Function rtc_count_is_syncing()

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

```
bool rtc_count_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 14-5. Return Values

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

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 as follows:

- 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.

Table 14-6. 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(void)
```

Function rtc_count_enable()

Enables the RTC module.

```
void rtc_count_enable(void)
```

Enables the RTC module once it has been configured, ready for use. Most module configuration parameters cannot be altered while the module is enabled.

Function rtc_count_disable()

```
void rtc_count_disable(void)
```

Disables the RTC module.

Function rtc_count_init()

Initializes the RTC module with given configurations.

```
enum status_code rtc_count_init(
  const struct rtc_count_config *const config)
```

Initializes the module, setting up all given configurations to provide the desired functionality of the RTC.

Table 14-7. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to the configuration structure.

Returns

Status of the initialization procedure.

Table 14-8. 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(
  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 14-9. Parameters

Data direction	Parameter name	Description
[in]	value	Ranging from -127 to 127 used for the correction.

Returns

Status of the calibration procedure.

Table 14-10. Return Values

Return value	Description
STATUS_OK	If calibration was executed correctly.

Return value	Description
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided.

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(
  const uint32_t count_value)
```

Sets the value of the counter to the specified value.

Table 14-11. Parameters

Data direction	Parameter name	Description
[in]	count_value	The value to be set in count register.

Returns Status of setting the register.

Table 14-12. 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(void)
```

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(
  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 14-13. Parameters

Data direction	Parameter name	Description
[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 14-14. 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(
  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 14-15. Parameters

Data direction	Parameter name	Description
[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 14-16. 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(
  uint16_t period_value)
```

Sets the given value to the period.

Note Only available in 16 bit mode.

Table 14-17. Parameters

Data direction	Parameter name	Description
[in]	period_value	The value to set to the period.

Returns Status of setting the period value.

Table 14-18. 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(
  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 14-19. Parameters

Data direction	Parameter name	Description
[out]	period_value	Pointer to value for return argument.

Returns Status of getting the period value.

Table 14-20. Return Values

Return value	Description
STATUS_OK	If the period value was read correctly.
STATUS_ERR_UNSUPPORTED_DEV	If incorrect mode was set.

Status management

Function rtc_count_is_overflow()

Check if an RTC overflow has occurred.

```
bool rtc_count_is_overflow(void)
```

Checks the overflow flag in the RTC. The flag is set when there is an overflow in the clock.

Returns Overflow state of the RTC module.

Table 14-21. 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(void)
```

Clears the RTC module counter overflow flag, so that new overflow conditions can be detected.

Function rtc_count_is_compare_match()

Check if RTC compare match has occurred.

```
bool rtc_count_is_compare_match(
  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 14-22. Parameters

Data direction	Parameter name	Description
[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(
  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 14-23. Parameters

Data direction	Parameter name	Description
[in]	comp_index	Index of compare to check current flag.

Returns Status indicating if flag was successfully cleared.

Table 14-24. 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.

Event management

Function rtc_count_enable_events()

Enables a RTC event output.

```
void rtc_count_enable_events(
  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 14-25. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to enable

Function rtc_count_disable_events()

Disables a RTC event output.

```
void rtc_count_disable_events(
  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 14-26. Parameters

Data direction	Parameter name	Description
[in]	events	Struct containing flags of events to disable

Callbacks

Function rtc_count_register_callback()

Registers callback for the specified callback type.

```
enum status_code rtc_count_register_callback(
  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 14-27. Parameters

Data direction	Parameter name	Description
[in]	callback	Pointer to the function desired for the specified callback
[in]	callback_type	Callback type to register

Returns Status of registering callback

Table 14-28. 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(
  enum rtc_count_callback callback_type)
```

When called, the currently registered callback for the given callback type will be removed.

Table 14-29. Parameters

Data direction	Parameter name	Description
[in]	callback_type	Specifies the callback type to unregister

Returns Status of unregistering callback

Table 14-30. 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(
  enum rtc_count_callback callback_type)
```

Enables the callback specified by the callback_type.

Table 14-31. Parameters

Data direction	Parameter name	Description
[in]	callback_type	Callback type to enable

Function rtc_count_disable_callback()

Disables callback.

```
void rtc_count_disable_callback(
  enum rtc_count_callback callback_type)
```

Disables the callback specified by the callback_type.

Table 14-32. Parameters

Data direction	Parameter name	Description
[in]	callback_type	Callback type to disable

14.7.3 Enumeration Definitions

Enum rtc_count_callback

The available callback types for the RTC count module.

Table 14-33. Members

Enum value	Description
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

Enum value	Description
RTC_COUNT_CALLBACK_OVERFLOW	Callback for overflow

Enum rtc_count_compare

Note Not all compare channels are available in all devices and modes.

Table 14-34. 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.

Enum rtc_count_mode

RTC Count operating modes, to select the counting width and associated module operation.

Table 14-35. 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.

Enum rtc_count_prescaler

The available input clock prescaler values for the RTC count module.

Table 14-36. Members

Enum value	Description
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.

Enum value	Description
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.

14.8 Extra Information for RTC (COUNT) Driver

14.8.1 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

14.8.2 Dependencies

This driver has the following dependencies:

- None

14.8.3 Errata

There are no errata related to this driver.

14.8.4 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

14.9 Examples for RTC (COUNT) Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 RTC Count Driver (RTC COUNT). 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 RTC (COUNT) - Basic
- Quick Start Guide for RTC (COUNT) - Callback

14.9.1 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 1000ms a LED on the board is toggled.

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`.

Setup

Initialization Code

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;
    config_rtc_count.continuously_update = true;
    config_rtc_count.compare_values[0]   = 1000;
    rtc_count_init(&config_rtc_count);

    rtc_count_enable();
}
```

Add to Main

Add the following to your `main()`.

```
configure_rtc_count();
```

Workflow

1. Create a 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.

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

```
rtc_count_get_config_defaults(&config_rtc_count);
```

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;
config_rtc_count.continuously_update = true;
config_rtc_count.compare_values[0]   = 1000;
```

4. Initialize the RTC module.

```
rtc_count_init(&config_rtc_count);
```

5. Enable the RTC module, so that it may begin counting.

```
rtc_count_enable();
```

Implementation

Code used to implement the initialized module.

Code

Add after initialization in main().

```
rtc_count_set_period(2000);

while (true) {
    if (rtc_count_is_compare_match(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_COUNT_COMPARE_0);
    }
}
```

Workflow

1. Set RTC period to 2000ms (2 seconds) so that it will overflow and reset back to zero every two seconds.

```
rtc_count_set_period(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_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_COUNT_COMPARE_0);
```

14.9.2 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.

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`.

Setup

Code

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;
    config_rtc_count.continuously_update = true;
    rtc_count_init(&config_rtc_count);

    rtc_count_enable();
}
```

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_overflow_callback, RTC_COUNT_CALLBACK_OVERFLOW);
    rtc_count_enable_callback(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(2000);
```

Workflow

1. Initialize system.

```
system_init();
```

2. Configure and enable module.

```
configure_rtc_count();
```

3. Create a RTC configuration structure to hold the desired RTC driver settings and fill it with the default driver configuration values.

Note

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

```
struct rtc_count_config config_rtc_count;
rtc_count_get_config_defaults(&config_rtc_count);
```

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;
config_rtc_count.continuously_update = true;
```

5. Initialize the RTC module.

```
rtc_count_init(&config_rtc_count);
```

6. Enable the RTC module, so that it may begin counting.

```
rtc_count_enable();
```

7. Configure callback functionality.

```
configure_rtc_callbacks();
```

   a. Register overflow callback.

   ```
   rtc_count_register_callback(
           rtc_overflow_callback, RTC_COUNT_CALLBACK_OVERFLOW);
   ```

   b. Enable overflow callback.

   ```
   rtc_count_enable_callback(RTC_COUNT_CALLBACK_OVERFLOW);
   ```

8. Set period.

```
rtc_count_set_period(2000);
```

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 */
   }
   ```

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);
   ```

15. SAM D20 Serial Peripheral Interface Driver (SERCOM SPI)

This driver for SAM D20 devices 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 peripherals are used by this module:

- SERCOM (Serial Communication Interface)

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.

15.2 Module Overview

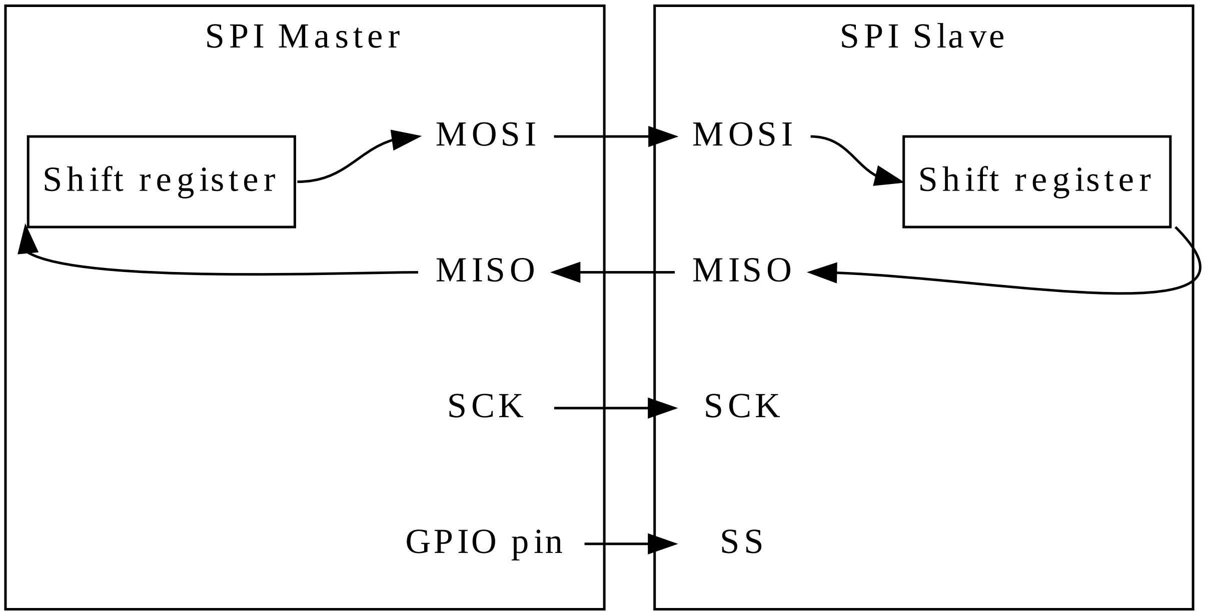
The Serial Peripheral Interface (SPI) is a high-speed synchronous data transfer interface using three or four pins. It allows fast communication between a master device and one or more peripheral devices.

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.

15.2.1 SPI Bus Connection

In Figure 15-1: SPI Bus Connection, the connection between one master and one slave is shown.

Figure 15-1. SPI Bus Connection



The different lines are as follows:

- **MOSI** Master Input, Slave Output. The line where the data is shifted out from the master and in to the slave.
- **MISO** Master Output Slave Input. The line where the data is shifted out from the slave and in to the master.
- **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 `Nth` slave connects its MISO back to the master. For a complete transaction, the master must shift `N+1` characters.

15.2.2 SPI Character Size

The SPI character size is configurable to 8 or 9 bits.

15.2.3 Master Mode

When configured as a master, the SS pin will be configured as an output.

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.

15.2.4 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.

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. When SS line is driven high, the slave will not receive any additional data.

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.

15.2.5 Data Modes

There are four combinations of SCK phase and polarity with respect to serial data. Table 15-1: SPI Data Modes 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 15-1. 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**
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

15.2.6 SERCOM Pads

The SERCOM pads are automatically configured as seen in Table 15-2: SERCOM SPI Pad Usages. 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 15-2. SERCOM SPI Pad Usages

Pin	Master SPI	Slave SPI
MOSI	**Output**	**Input**
MISO	**Input**	**Output**
SCK	**Output**	**Input**
SS	**User defined output enable**	**Input**
MOSI	Output	Input
MISO	Input	Output
SCK	Output	Input
SS	User defined output enable	Input

For SERCOM pad multiplexer position documentation, see Mux Settings.

15.2.7 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.

run_in_standby	Slave	Master
false	**Disabled, all reception is dropped**	**GCLK disabled when master is idle, wake on transmit complete**
true	**Wake on reception**	**GCLK is enabled while in sleep modes, wake on all interrupts**
false	Disabled, all reception is dropped	GCLK disabled when master is idle, wake on transmit complete
true	Wake on reception	GCLK is enabled while in sleep modes, wake on all interrupts

15.2.8 Clock Generation

In SPI master mode, the clock (SCK) is generated internally using the SERCOM baud rate 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.

15.3 Special Considerations

15.3.1 Pin MUX Settings

The pin MUX settings must be configured properly, as not all settings can be used in different modes of operation.

15.4 Extra Information

For extra information see Extra Information for SERCOM SPI Driver. This includes:

- Acronyms
- Dependencies
- Workarounds Implemented by Driver
- Module History

15.5 Examples

For a list of examples related to this driver, see Examples for SERCOM SPI Driver.

15.6 API Overview

15.6.1 Variable and Type Definitions

Type spi_callback_t

```
typedef void(* spi_callback_t )(const struct spi_module *const module)
```

Type of the callback functions

15.6.2 Structure Definitions

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 15-3. Members

Type	Name	Description
union spi_config.@138	@138	Union for slave or master specific configuration Union for slave or master specific configuration
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.
enum spi_mode	mode	SPI mode
enum spi_signal_mux_setting	mux_setting	Mux setting
uint32_t	pinmux_pad0	PAD0 pinmux
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
enum spi_transfer_mode	transfer_mode	Transfer mode

Union spi_config.__unnamed__

Union for slave or master specific configuration

Table 15-4. Members

Type	Name	Description
struct spi_master_config	master	Master specific configuration
struct spi_slave_config	slave	Slave specific configuration

Struct spi_master_config

SPI Master configuration structure

Table 15-5. Members

Type	Name	Description
uint32_t	baudrate	Baud rate

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.

Struct spi_slave_config

SPI slave configuration structure

Table 15-6. Members

Type	Name	Description
uint8_t	address	Address
uint8_t	address_mask	Address mask
enum spi_addr_mode	address_mode	Address mode
enum spi_frame_format	frame_format	Frame format
bool	preload_enable	Preload data to the shift register while SS is high

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 15-7. 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

Struct spi_slave_inst_config

SPI Peripheral slave configuration structure

Table 15-8. 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

15.6.3 Macro Definitions

Macro PINMUX_DEFAULT

```
#define PINMUX_DEFAULT 0
```

Macro PINMUX_UNUSED

```
#define PINMUX_UNUSED 0xFFFFFFFF
```

Macro SPI_TIMEOUT

```
#define SPI_TIMEOUT 10000
```

15.6.4 Function Definitions

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 15-9. 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 15-10. 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 a 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 15-11. 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 15-12. Parameters

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

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 15-13. Parameters

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

Data direction	Parameter name	Description
[out]	tx_data	Pointer to data buffer to receive
[in]	length	Data buffer length

Returns Status of the write request operation.

Table 15-14. 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 15-15. 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 15-16. 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 tranfer 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 15-17. 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 15-18. 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,
  enum spi_job_type job_type)
```

This function will abort the specified job type.

Table 15-19. Parameters

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

Data direction	Parameter name	Description
[in]	job_type	Type of job to abort

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,
  enum spi_job_type job_type)
```

Retrieves the current statue of a job that was previously issued.

Table 15-20. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	job_type	Type of job to check

Returns Current job status

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 8 bit
- Not enabled in sleep mode
- Receiver enabled
- Baudrate 100000
- Default pinmux settings for all pads
- GCLK generator 0

Table 15-21. 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 15-22. 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,
  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 15-23. 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 15-24. 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 15-25. 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.

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 15-26. Parameters

Data direction	Parameter name	Description
[inout]	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 15-27. Parameters

Data direction	Parameter name	Description
[inout]	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 15-28. Parameters

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

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 15-29. Parameters

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

Returns Indication of whether any writes are ongoing

Table 15-30. 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 15-31. 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 15-32. 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 15-33. 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 15-34. 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

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 15-35. 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 15-36. 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 15-37. 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 15-38. 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.

Receiver must be enabled in the configuration

Table 15-39. 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 15-40. 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 15-41. Parameters

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

Data direction	Parameter name	Description
[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 15-42. Return Values

Return value	Description
STATUS_OK	If the read 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.
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.

If address matching is enabled for the slave, the first character received and placed in the buffer will be the address.

Table 15-43. 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 15-44. 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 15-45. 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 15-46. 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 15-47. 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 15-48. 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

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 15-49. Parameters

Data direction	Parameter name	Description
[in]	module	SPI hardware module

Returns Synchronization status of the underlying hardware module

Table 15-50. Return Values

Return value	Description
true	Module synchronization is ongoing
false	Module synchronization is not ongoing

15.6.5 Enumeration Definitions

Enum spi_addr_mode

For slave mode when using the SPI frame with address format.

Table 15-51. 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 in the spi_config struct.

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 15-52. 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 entire buffer was read or written from slave

Enum spi_character_size

Table 15-53. Members

Enum value	Description
SPI_CHARACTER_SIZE_8BIT	8 bit character
SPI_CHARACTER_SIZE_9BIT	9 bit character

Enum spi_data_order

Table 15-54. 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

Enum spi_frame_format

Frame format for slave mode.

Table 15-55. Members

Enum value	Description
SPI_FRAME_FORMAT_SPI_FRAME	SPI frame
SPI_FRAME_FORMAT_SPI_FRAME_ADDR	SPI frame with address

Enum spi_interrupt_flag

Interrupt flags for the SPI module

Table 15-56. 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

Enum spi_job_type

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

Table 15-57. Members

Enum value	Description
SPI_JOB_READ_BUFFER	Asynchronous SPI read into a user provided buffer
SPI_JOB_WRITE_BUFFER	Asynchronous SPI write from a user provided buffer
SPI_JOB_TRANSCEIVE_BUFFER	Asynchronous SPI transceive from user provided buffers

Enum spi_mode

Table 15-58. Members

Enum value	Description
SPI_MODE_MASTER	Master mode
SPI_MODE_SLAVE	Slave mode

Enum spi_signal_mux_setting

Set the functionality of the SERCOM pins. As not all settings can be used in different modes of operation, proper settings must be chosen according to the rest of the configuration.

Table 15-59. Members

Enum value	Description
SPI_SIGNAL_MUX_SETTING_A	See Mux Setting A
SPI_SIGNAL_MUX_SETTING_B	See Mux Setting B

Enum value	Description
SPI_SIGNAL_MUX_SETTING_C	See Mux Setting C
SPI_SIGNAL_MUX_SETTING_D	See Mux Setting D
SPI_SIGNAL_MUX_SETTING_E	See Mux Setting E
SPI_SIGNAL_MUX_SETTING_F	See Mux Setting F
SPI_SIGNAL_MUX_SETTING_G	See Mux Setting G
SPI_SIGNAL_MUX_SETTING_H	See Mux Setting H

Enum spi_transfer_mode

SPI transfer mode.

Table 15-60. 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

15.7 Mux Settings

The different options for functionality of the SERCOM pads. As not all settings can be used in different modes of operation, proper settings must be chosen according to the rest of the configuration.

Pin	Master Description	Slave Description
DO	MOSI	MISO
DI	MISO	MOSI
SLAVE_SS	None	Slave Select
SCK	Serial Clock	Serial Clock

15.7.1 Mux Setting A

- Master mode: Receiver turned off
- Slave mode: Receiver turned off
- Enum: SPI_SIGNAL_MUX_SETTING_A

Function	Pad0	Pad1	Pad2	Pad3
SCK		**x**		
SLAVE_SS			**x**	
DO	**x**			
DI	**x**			
SCK		x		
SLAVE_SS			x	

Function	Pad0	Pad1	Pad2	Pad3
SCK		**x**		
SLAVE_SS			**x**	
DO	**x**			
DI	**x**			
DO	x			
DI	x			

15.7.2 Mux Setting B

- Master mode: Receiver turned off
- Slave mode: Not applicable
- Enum: SPI_SIGNAL_MUX_SETTING_B

Function	Pad0	Pad1	Pad2	Pad3
SCK		**x**		
SLAVE_SS			**x**	
DO	**x**			
DI		**x**		
SCK		x		
SLAVE_SS			x	
DO	x			
DI		x		

15.7.3 Mux Setting C

- Master mode: No restrictions
- Slave mode: Not applicable
- Enum: SPI_SIGNAL_MUX_SETTING_C

Function	Pad0	Pad1	Pad2	Pad3
SCK		**x**		
SLAVE_SS			**x**	
DO	**x**			
DI			**x**	
SCK		x		
SLAVE_SS			x	
DO	x			
DI			x	

15.7.4 Mux Setting D

- Master mode: No restrictions
- Slave mode: No restrictions
- Enum: SPI_SIGNAL_MUX_SETTING_D

Function	Pad0	Pad1	Pad2	Pad3
SCK		**x**		
SLAVE_SS			**x**	
DO	**x**			
DI				**x**
SCK		x		
SLAVE_SS			x	
DO	x			
DI				x

15.7.5 Mux Setting E

- Master mode: No restrictions
- Slave mode: No restrictions
- Enum: SPI_SIGNAL_MUX_SETTING_E

Function	Pad0	Pad1	Pad2	Pad3
SCK				**x**
SLAVE_SS		**x**		
DO			**x**	
DI	**x**			
SCK				x
SLAVE_SS		x		
DO			x	
DI	x			

15.7.6 Mux Setting F

- Master mode: No restrictions
- Slave mode: Not applicable
- Enum: SPI_SIGNAL_MUX_SETTING_F

Function	Pad0	Pad1	Pad2	Pad3
SCK				**x**
SLAVE_SS		**x**		
DO			**x**	
DI		**x**		
SCK				x
SLAVE_SS		x		
DO			x	
DI		x		

15.7.7 Mux Setting G

- Master mode: Receiver turned off

- Slave mode: Receiver turned off
- Enum: SPI_SIGNAL_MUX_SETTING_G

Function	Pad0	Pad1	Pad2	Pad3
SCK				**x**
SLAVE_SS		**x**		
DO			**x**	
DI			**x**	
SCK				x
SLAVE_SS		x		
DO			x	
DI			x	

15.7.8 Mux Setting H

- Master mode: Receiver turned off
- Slave mode: Not applicable
- Enum: SPI_SIGNAL_MUX_SETTING_H

Function	Pad0	Pad1	Pad2	Pad3
SCK				**x**
SLAVE_SS		**x**		
DO			**x**	
DI				**x**
SCK				x
SLAVE_SS		x		
DO			x	
DI				x

15.8 Extra Information for SERCOM SPI Driver

15.8.1 Acronyms

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

Acronym	Description
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

15.8.2 Dependencies

The SPI driver has the following dependencies:

- System Pin Multiplexer Driver

15.8.3 Workarounds Implemented by Driver

No workarounds in 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
Initial Release

15.9 Examples for SERCOM SPI Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Serial Peripheral Interface Driver (SERCOM SPI). 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 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

15.9.1 Quick Start Guide for SERCOM SPI Master - Polled

In this use case, the SPI on extension header 1 of the Xplained Pro board will configured with the following settings:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- Mux Setting E
  - MOSI on pad 2, extension header 1, pin 16
  - MISO on pad 0, extension header 1, pin 17
  - SCK on pad 3, extension header 1, pin 18
  - SS on extension header 1, pin 15
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

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 const 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();
```

Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Setup the SPI:

   ```
   configure_spi_master();
   ```

   a. Create configuration struct.

      ```
      struct spi_config config_spi_master;
      ```

   b. Create peripheral slave configuration struct.

      ```
      struct spi_slave_inst_config slave_dev_config;
      ```

   c. Create peripheral slave software device instance struct.

      ```
      struct spi_slave_inst slave;
      ```

   d. Get default peripheral slave configuration.

      ```
      spi_slave_inst_get_config_defaults(&slave_dev_config);
      ```

   e. Set Slave Select pin.

      ```
      slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
      ```

   f. Initialize peripheral slave software instance with configuration.

      ```
      spi_attach_slave(&slave, &slave_dev_config);
      ```

   g. Get default configuration to edit.

      ```
      spi_get_config_defaults(&config_spi_master);
      ```

   h. Set mux setting E.

      ```
      config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
      ```

   i. Set pinmux for pad 0 (data in (MISO) on extension header 1, pin 17).

      ```
      config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
      ```

   j. Set pinmux for pad 1 as unused, so the pin can be used for other purposes.

```
config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
```

k. Set pinmux for pad 2 (data out (MOSI) on extension header 1, pin 16).

```
config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

l. Set pinmux for pad 3 (SCK on extension header 1, pin 18).

```
config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

m. Initialize SPI module with configuration.

```
spi_init(&spi_master_instance, EXT1_SPI_MODULE, &config_spi_master);
```

n. Enable SPI module.

```
spi_enable(&spi_master_instance);
```

Use Case

Code

Add the following to your user application `main()`:

```
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);

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

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. Infinite loop.

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

15.9.2 Quick Start Guide for SERCOM SPI Slave - Polled

In this use case, the SPI on extension header 1 of the Xplained Pro board will configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- Mux Setting E
    - MISO on pad 2, extension header 1, pin 16
    - MOSI on pad 0, extension header 1, pin 17
    - SCK on pad 3, extension header 1, pin 18
    - SS on pad 1, extension header 1, pin
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

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 const 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
```

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.slave.preload_enable = true;
    config_spi_slave.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():

```
/* Initialize system */
system_init();

configure_spi_slave();
```

Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Setup the SPI:

   ```
   configure_spi_slave();
   ```

   a. Create configuration struct.

   ```
   struct spi_config config_spi_slave;
   ```

   b. Get default configuration to edit.

   ```
   spi_get_config_defaults(&config_spi_slave);
   ```

   c. Set the SPI in slave mode.

   ```
   ```

   d. Enable preloading of shift register.

   ```
   config_spi_slave.slave.preload_enable = true;
   ```

e. Set frame format to SPI frame.

```
config_spi_slave.slave.preload_enable = true;
```

f. Set mux setting E.

```
config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

g. Set pinmux for pad 0 (data in (MOSI) on extension header 1, pin 17).

```
config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

h. Set pinmux for pad 1 (slave select on on extension header 1, pin 15)

```
config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
```

i. Set pinmux for pad 2 (data out (MISO) on extension header 1, pin 16).

```
config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

j. Set pinmux for pad 3 (SCK on extension header 1, pin 18).

```
config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

k. Initialize SPI module with configuration.

```
spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);
```

l. Enable SPI module.

```
spi_enable(&spi_slave_instance);
```

Use Case

Code

Add the following to your user application `main()`:

```
while (spi_write_buffer_wait(&spi_slave_instance, buffer, BUF_LENGTH !=
        STATUS_OK)) {
    /* Wait for transfer from master */
}


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

Workflow

1. Write buffer to SPI master. Placed in a loop to retry in case of a timeout before a master initates a transaction.

```
while (spi_write_buffer_wait(&spi_slave_instance, buffer, BUF_LENGTH !=
        STATUS_OK)) {
    /* Wait for transfer from master */
}
```

2. Infinite loop.

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

15.9.3 Quick Start Guide for SERCOM SPI Master - Callback

In this use case, the SPI on extension header 1 of the Xplained Pro board will configured with the following settings:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- Mux Setting E
  - MOSI on pad 2, extension header 1, pin 16
  - MISO on pad 0, extension header 1, pin 17
  - SCK on pad 3, extension header 1, pin 18
  - SS on extension header 1, pin 15
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

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);

}
```

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_TRANSMITTED);
    spi_enable_callback(&spi_master_instance, SPI_CALLBACK_BUFFER_TRANSMITTED);
}
```

A global variable that can flag to the application that the buffer has been transferred:

```
volatile bool transfer_complete_spi_master = false;
```

Callback function:

```
static void callback_spi_master(const struct spi_module *const module)
{
    transfer_complete_spi_master = true;
}
```

Add to user application main():

```
/* Initialize system */
system_init();

configure_spi_master();
configure_spi_master_callbacks();
```

Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Setup the SPI:

   ```
   configure_spi_master();
   ```

   a. Create configuration struct.

      ```
      struct spi_config config_spi_master;
      ```

   b. Create peripheral slave configuration struct.

      ```
      struct spi_slave_inst_config slave_dev_config;
      ```

   c. Get default peripheral slave configuration.

      ```
      spi_slave_inst_get_config_defaults(&slave_dev_config);
      ```

   d. Set Slave Select pin.

      ```
      slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
      ```

   e. Initialize peripheral slave software instance with configuration.

      ```
      spi_attach_slave(&slave, &slave_dev_config);
      ```

   f. Get default configuration to edit.

      ```
      spi_get_config_defaults(&config_spi_master);
      ```

   g. Set mux setting E.

      ```
      config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
      ```

   h. Set pinmux for pad 0 (data in (MISO) on extension header 1, pin 17).

      ```
      config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
      ```

   i. Set pinmux for pad 1 as unused, so the pin can be used for other purposes.

```
config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
```

j. Set pinmux for pad 2 (data out (MOSI) on extension header 1, pin 16).

```
config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

k. Set pinmux for pad 3 (SCK on extension header 1, pin 18).

```
config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

l. Initialize SPI module with configuration.

```
spi_init(&spi_master_instance, EXT1_SPI_MODULE, &config_spi_master);
```

m. Enable SPI module.

```
spi_enable(&spi_master_instance);
```

3. Setup the callback functionality:

```
configure_spi_master_callbacks();
```

a. Register callback function for buffer transmitted

```
spi_register_callback(&spi_master_instance, callback_spi_master,
        SPI_CALLBACK_BUFFER_TRANSMITTED);
```

b. Enable callback for buffer transmitted

```
spi_enable_callback(&spi_master_instance, SPI_CALLBACK_BUFFER_TRANSMITTED);
```

Use Case

Code

Add the following to your user application `main()`:

```
spi_select_slave(&spi_master_instance, &slave, true);
spi_write_buffer_job(&spi_master_instance, buffer, BUF_LENGTH);
while (!transfer_complete_spi_master) {
    /* Wait for write complete */
}
spi_select_slave(&spi_master_instance, &slave, false);

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

Workflow

1. Select slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Write buffer to SPI slave.

```
spi_write_buffer_job(&spi_master_instance, buffer, BUF_LENGTH);
```

3. Wait for the transfer to be complete.

```
while (!transfer_complete_spi_master) {
    /* Wait for write complete */
}
```

4. Deselect slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

5. Infinite loop.

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

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.

```
transfer_complete_spi_master = true;
```

15.9.4 Quick Start Guide for SERCOM SPI Slave - Callback

In this use case, the SPI on extension header 1 of the Xplained Pro board will configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- Mux Setting E
    - MISO on pad 2, extension header 1, pin 16
    - MOSI on pad 0, extension header 1, pin 17
    - SCK on pad 3, extension header 1, pin 18
    - SS on pad 1, extension header 1, pin 15

- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

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[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.slave.preload_enable = true;
    config_spi_slave.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_TRANSMITTED);
    spi_enable_callback(&spi_slave_instance, SPI_CALLBACK_BUFFER_TRANSMITTED);
}
```

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(const struct spi_module *const module)
{
    transfer_complete_spi_slave = true;
}
```

Add to user application main():

```
/* Initialize system */
system_init();

configure_spi_slave();
configure_spi_slave_callbacks();
```

Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Setup the SPI:

   ```
   configure_spi_slave();
   ```

   a. Create configuration struct.

      ```
      struct spi_config config_spi_slave;
      ```

   b. Get default configuration to edit.

      ```
      spi_get_config_defaults(&config_spi_slave);
      ```

   c. Set the SPI in slave mode.

      ```
      ```

   d. Enable preloading of shift register.

      ```
      config_spi_slave.slave.preload_enable = true;
      ```

e. Set frame format to SPI frame.

```
config_spi_slave.slave.preload_enable = true;
```

f. Set mux setting E.

```
config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

g. Set pinmux for pad 0 (data in (MOSI) on extension header 1, pin 17).

```
config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

h. Set pinmux for pad 1 (slave select on on extension header 1, pin 15)

```
config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
```

i. Set pinmux for pad 2 (data out (MISO) on extension header 1, pin 16).

```
config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

j. Set pinmux for pad 3 (SCK on extension header 1, pin 18).

```
config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

k. Initialize SPI module with configuration.

```
spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);
```

l. Enable SPI module.

```
spi_enable(&spi_slave_instance);
```

3. Setup the callback functionality:

```
configure_spi_slave_callbacks();
```

a. Register callback function for buffer transmitted

```
spi_register_callback(&spi_slave_instance, spi_slave_callback,
        SPI_CALLBACK_BUFFER_TRANSMITTED);
```

b. Enable callback for buffer transmitted

```
spi_enable_callback(&spi_slave_instance, SPI_CALLBACK_BUFFER_TRANSMITTED);
```

Use Case

Code

Add the following to your user application `main()`:

```
spi_write_buffer_job(&spi_slave_instance, buffer, BUF_LENGTH);
while(!transfer_complete_spi_slave) {
    /* Wait for transfer from master */
}


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

Workflow

1. Initiate a write buffer job.

   ```
   spi_write_buffer_job(&spi_slave_instance, buffer, BUF_LENGTH);
   ```

2. Wait for the transfer to be complete.

   ```
   while(!transfer_complete_spi_slave) {
       /* Wait for transfer from master */
   }
   ```

3. Infinite loop.

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

Callback

When the buffer is successfully transmitted to 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;
   ```

16. SAM D20 Serial USART Driver (SERCOM USART)

This driver for SAM D20 devices 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 peripherals are used by this module:

- SERCOM (Serial Communication Interface)

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 interfaces on the system and configure it to run as a USART interface in either synchronous or asynchronous mode.

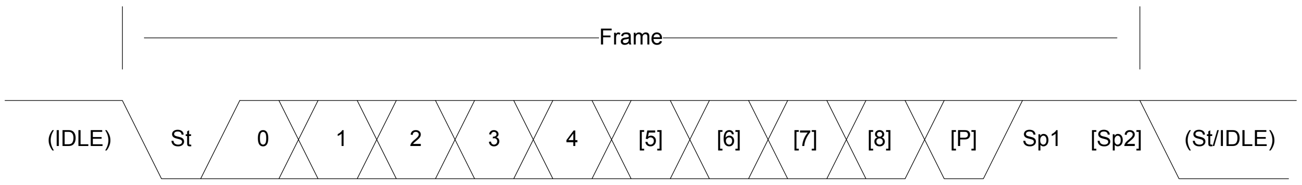
16.2.1 Frame Format

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: USART Frame overview. Table 16-1: USART Frame Parameters 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**
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



16.2.2 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)

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.

16.2.3 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)

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 baud rate closely enough the receiver will be unable to synchronize the frame(s), and garbage transmissions will result.

16.2.4 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.

16.2.5 GPIO configuration

the SERCOM module have four internal PADS where the RX pin can be placed at all the PADS, and the TX and XCK pins have two predefined positions that can be changed. The PADS can then be routed to an external GPIO pin using the normal pin multiplexing scheme on the SAM D20.

For SERCOM pad multiplexer position documentation, see SERCOM USART MUX Settings.

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 signalling 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

Type usart_callback_t

```
typedef void(* usart_callback_t )(const struct usart_module *const module)
```

16.6.2 Structure Definitions

Struct usart_config

Configuration options for USART

Table 16-2. Members

Type	Name	Description
uint32_t	baudrate	USART baud rate
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.
enum usart_dataorder	data_order	USART bit order (MSB or LSB first)
uint32_t	ext_clock_freq	External clock frequency in synchronous mode. This must be set if use_external_clock is true.
enum gclk_generator	generator_source	GCLK generator source
enum usart_signal_mux_settings	mux_setting	USART pin out
enum usart_parity	parity	USART parity
uint32_t	pinmux_pad0	PAD0 pinmux
uint32_t	pinmux_pad1	PAD1 pinmux

Type	Name	Description
uint32_t	pinmux_pad2	PAD2 pinmux
uint32_t	pinmux_pad3	PAD3 pinmux
bool	run_in_standby	If true the USART will be kept running in Standby sleep mode
enum usart_stopbits	stopbits	Number of stop bits
enum usart_transfer_mode	transfer_mode	USART in asynchronous or synchronous mode
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.

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

Macro PINMUX_DEFAULT

```
#define PINMUX_DEFAULT 0
```

Macro PINMUX_UNUSED

```
#define PINMUX_UNUSED 0xFFFFFFFF
```

Macro USART_TIMEOUT

```
#define USART_TIMEOUT 0xFFFF
```

16.6.4 Function Definitions

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 16-3. 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 16-4. Parameters

Data direction	Parameter name	Description
[inout]	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 16-5. Parameters

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

Data direction	Parameter name	Description
[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 16-6. Parameters

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

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 16-7. 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 16-8. 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.

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 16-9. 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 16-10. 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 16-11. 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

Returns Status of the operation

Table 16-12. Return Values

Return value	Description
STATUS_OK	If operation was completed successfully.

Return value	Description
STATUS_BUSY	If operation was not completed,

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 16-13. 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

Returns Status of the operation

Table 16-14. Return Values

Return value	Description
STATUS_OK	If operation was completed.
STATUS_BUSY	If operation was not completed,

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 16-15. 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 16-16. 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 16-17. 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.

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 16-18. 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 16-19. Return Values

Return value	Description
STATUS_OK	If the operation was completed

Return value	Description
STATUS_BUSY	If the operation was not completed, due to the USART module being busy.

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 16-20. 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 16-21. 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 baud rate 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

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 16-22. 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

Returns Status of the operation

Table 16-23. 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

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 16-24. 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

Returns Status of the operation.

Table 16-25. Return Values

Return value	Description
STATUS_OK	If operation was completed

Return value	Description
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 baud rate 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

Enabling/Disabling receiver and transmitter

Function usart_enable_transceiver()

Enable Transceiver.

```
void usart_enable_transceiver(
  const struct usart_module *const module,
  enum usart_transceiver_type transceiver_type)
```

Enable the given transceiver. Either RX or TX.

Table 16-26. 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(
  const struct usart_module *const module,
  enum usart_transceiver_type transceiver_type)
```

Disable the given transceiver (RX or TX).

Table 16-27. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transceiver type.

Function usart_disable()

Disable module.

```
void usart_disable(
  const struct usart_module *const module)
```

Disables the USART module

Table 16-28. Parameters

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

Function usart_enable()

Enable the module.

```
void usart_enable(
  const struct usart_module *const module)
```

Enables the USART module

Table 16-29. Parameters

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

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
- 1 stop bit
- 9600 baud
- GCLK generator 0 as clock source
- Default pin configuration

The configuration struct will be updated with the default configuration.

Table 16-30. Parameters

Data direction	Parameter name	Description
[inout]	config	Pointer to configuration struct

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 16-31. 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 16-32. Return Values

Return value	Description
STATUS_OK	The initialization was successful
STATUS_BUSY	The USART module is busy resetting
STATUS_ERR_DENIED	The USART have 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

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 of any new actions until sync is complete. If this functions is not run; the functions will block until the sync has completed.

Table 16-33. Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to peripheral module

Returns Peripheral sync status

Table 16-34. 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.

Function usart_reset()

Resets the USART module.

```
void usart_reset(
  const struct usart_module *const module)
```

Disables and resets the USART module.

Table 16-35. Parameters

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

16.6.5 Enumeration Definitions

Enum usart_callback

Callbacks for the Asynchronous USART driver

Table 16-36. 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

Enum usart_character_size

Number of bits for the character sent in a frame.

Table 16-37. Members

Enum value	Description
USART_CHARACTER_SIZE_5BIT	The char being sent in a frame is 5 bits long
USART_CHARACTER_SIZE_6BIT	The char being sent in a frame is 6 bits long
USART_CHARACTER_SIZE_7BIT	The char being sent in a frame is 7 bits long
USART_CHARACTER_SIZE_8BIT	The char being sent in a frame is 8 bits long
USART_CHARACTER_SIZE_9BIT	The char being sent in a frame is 9 bits long

Enum usart_dataorder

The data order decides which of MSB or LSB is shifted out first when data is transferred

Table 16-38. 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

Enum usart_parity

Select parity USART parity mode

Table 16-39. 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

Enum usart_signal_mux_settings

Set the functionality of the SERCOM pins.

Table 16-40. Members

Enum value	Description
USART_RX_0_TX_0_XCK_1	See MUX Setting A
USART_RX_0_TX_2_XCK_3	See MUX Setting B
USART_RX_1_TX_0_XCK_1	See MUX Setting C
USART_RX_1_TX_2_XCK_3	See MUX Setting D
USART_RX_2_TX_0_XCK_1	See MUX Setting E
USART_RX_2_TX_2_XCK_3	See MUX Setting F
USART_RX_3_TX_0_XCK_1	See MUX Setting G
USART_RX_3_TX_2_XCK_3	See MUX Setting H

Enum usart_stopbits

Number of stop bits for a frame.

Table 16-41. Members

Enum value	Description
USART_STOPBITS_1	Each transferred frame contains 1 stop bit
USART_STOPBITS_2	Each transferred frame contains 2 stop bits

Enum usart_transceiver_type

Select Receiver or Transmitter

Table 16-42. Members

Enum value	Description
USART_TRANSCEIVER_RX	The parameter is for the Receiver
USART_TRANSCEIVER_TX	The parameter is for the Transmitter

Enum usart_transfer_mode

Select USART transfer mode

Table 16-43. Members

Enum value	Description
USART_TRANSFER_SYNCHRONOUSLY	Transfer of data is done synchronously
USART_TRANSFER_ASYNCHRONOUSLY	Transfer of data is done asynchronously

16.7 SERCOM USART MUX Settings

The different options for functionality of the SERCOM pads.

16.7.1 MUX Setting A

Enum: USART_RX_0_TX_0_XCK_1 [329]

Function	RX	TX	XCK
PAD0	**x**	**x**	
PAD1			**x**
PAD2			
PAD3			
PAD0	x	x	
PAD1			x
PAD2			
PAD3			

16.7.2 MUX Setting B

Enum: USART_RX_0_TX_2_XCK_3 [329]

Function	RX	TX	XCK
PAD0	**x**		
PAD1			
PAD2		**x**	
PAD3			**x**
PAD0	x		

Function	RX	TX	XCK
PAD0	x		
PAD1			
PAD2		x	
PAD3			x
PAD1			
PAD2		x	
PAD3			x

16.7.3 MUX Setting C

Enum: USART_RX_1_TX_0_XCK_1 [329]

Function	RX	TX	XCK
PAD0		x	
PAD1	x		x
PAD2			
PAD3			
PAD0		x	
PAD1	x		x
PAD2			
PAD3			

16.7.4 MUX Setting D

Enum: USART_RX_1_TX_2_XCK_3 [329]

Function	RX	TX	XCK
PAD0			
PAD1	x		
PAD2		x	
PAD3			x
PAD0			
PAD1	x		
PAD2		x	
PAD3			x

16.7.5 MUX Setting E

Enum: USART_RX_2_TX_0_XCK_1 [329]

Function	RX	TX	XCK
PAD0		x	
PAD1			x
PAD2	x		
PAD3			
PAD0		x	

Function	RX	TX	XCK
PAD0		**x**	
PAD1			**x**
PAD2	**x**		
PAD3			
PAD1			x
PAD2	x		
PAD3			

16.7.6 MUX Setting F

Enum: USART_RX_2_TX_2_XCK_3 [329]

Function	RX	TX	XCK
PAD0			
PAD1			
PAD2	**x**	**x**	
PAD3			**x**
PAD0			
PAD1			
PAD2	x	x	
PAD3			x

16.7.7 MUX Setting G

Enum: USART_RX_3_TX_0_XCK_1 [329]

Function	RX	TX	XCK
PAD0		**x**	
PAD1			**x**
PAD2			
PAD3	**x**		
PAD0		x	
PAD1			x
PAD2			
PAD3	x		

16.7.8 MUX Setting H

Enum: USART_RX_3_TX_2_XCK_3 [329]

Function	RX	TX	XCK
PAD0			
PAD1			
PAD2		**x**	
PAD3	**x**		**x**
PAD0			

Function	RX	TX	XCK
PAD0			
PAD1			
PAD2		**x**	
PAD3	**x**		**x**
PAD1			
PAD2		x	
PAD3	x		x

16.8 Extra Information for SERCOM USART Driver

16.8.1 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

16.8.2 Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver
- System clock configuration

16.8.3 Errata

There are no errata related to this driver.

16.8.4 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

16.9 Examples for SERCOM USART Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Serial USART Driver (SERCOM USART). 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 SERCOM USART - Basic

- Quick Start Guide for SERCOM USART - Callback

16.9.1 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 1 Stop Bit
- TX and RX connected to the Xplained PRO Embedded Debugger virtual COM port

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    = 57600;
    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);

    usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_TX);
    usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_RX);
}
```

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.

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

```
struct usart_module usart_instance;
```

2. Configure the USART module.

   a. 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;
   ```

   b. Initialize the USART configuration struct with the module's default values.

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

   ```
   usart_get_config_defaults(&config_usart);
   ```

   c. Alter the USART settings to configure the physical pinout, baud rate and other relevant parameters.

   ```
   config_usart.baudrate    = 57600;
   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;
   ```

   d. 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) {
   }
   ```

   e. Enable the USART module so that the transceivers can be configured.

   ```
   usart_enable(&usart_instance);
   ```

3. Enable the RX and TX transceivers for bidirectional USART communications.

```
usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_TX);
usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_RX);
```

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) {
}
```

16.9.2 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 1 Stop Bit
- TX and RX connected to the Xplained PRO Embedded Debugger virtual COM port

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(const struct usart_module *const usart_module)
{
    usart_write_buffer_job(&usart_instance,
            (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);
}

void usart_write_callback(const 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);

    usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_TX);
    usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_RX);
}

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.

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

```
struct usart_module usart_instance;
```

2. Configure the USART module.

   a. 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;
   ```

   b. Initialize the USART configuration struct with the module's default values.

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

   ```
   usart_get_config_defaults(&config_usart);
   ```

   c. Alter the USART settings to configure the physical pinout, baud rate 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;
   ```

   d. 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) {
   }
   ```

   e. Enable the USART module so that the transceivers can be configured.

   ```
   usart_enable(&usart_instance);
   ```

3. Enable the RX and TX transceivers for bidirectional USART communications.

```
usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_TX);
usart_enable_transceiver(&usart_instance, USART_TRANSCEIVER_RX);
```

4. Configure the USART callbacks.

    a. 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);
    ```

    b. 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);
    ```

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_job(&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_job(&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);
```

17. SAM D20 System Driver (SYSTEM)

This driver for SAM D20 devices 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:

- SYSCTRL (System Control)
- PM (Power Manager)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for SYSTEM
- Examples
- API Overview

17.1 Prerequisites

There are no prerequisites for this module.

17.2 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)

17.2.1 Voltage References

The various analog modules within the SAM D20 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 D20 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.

17.2.2 System Reset Cause

In some application 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.

17.2.3 Sleep Modes

The SAM D20 devices have several sleep modes, where 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 17-1: SAM D20 Device Sleep Modes lists the clock settings of the different sleep modes.

Table 17-1. SAM D20 Device Sleep Modes

Sleep mode	CPU clock	AHB clock	APB clocks	Clock sources	System clock	32KHz	Reg mode	RAM mode
IDLE 0	Stop	Run	Run	Run	Run	Run	Normal	Normal
IDLE 1	Stop	Stop	Run	Run	Run	Run	Normal	Normal
IDLE 2	Stop	Stop	Stop	Run	Run	Run	Normal	Normal
STANDBY	Stop	Stop	Stop	Stop	Stop	Stop	Low Power	Source/ Drain biasing

To enter device sleep, one of the available sleep modes must be set, and the function to enter sleep called. The device will automatically wake up in response to an interrupt being generated or other device event.

Some peripheral clocks will remain enabled during sleep, depending on their configuration; if desired, modules can remain clocked during sleep to allow them to continue to operate while other parts of the system are powered down to save power.

17.3 Special Considerations

Most of the functions in this driver have device specific restrictions and caveats; refer to your device datasheet.

17.4 Extra Information for SYSTEM

For extra information see Extra Information for SYSTEM Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

17.5 Examples

For SYSTEM module related examples, please refer to the sub-modules listed in the system module overview.

17.6 API Overview

17.6.1 Function Definitions

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.

Voltage references

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 17-2. 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 17-3. Parameters

Data direction	Parameter name	Description
[in]	vref	Voltage reference to disable

Device sleep

Function system_set_sleepmode()

Set the sleep mode of the device.

```
enum status_code 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 17-4. Parameters

Data direction	Parameter name	Description
[in]	sleep_mode	Sleep mode to configure for the next sleep operation

Table 17-5. Return Values

Return value	Description
STATUS_OK	Operation completed successfully
STATUS_ERR_INVALID_ARG	The requested sleep mode was invalid or not available

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, then a WFI (Wait For Interrupt) instruction to place the device into the sleep mode specified by system_set_sleepmode until woken by an interrupt.

Reset cause

Function system_get_reset_cause()

Return 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.

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)

17.6.2 Enumeration Definitions

Enum system_reset_cause

List of possible reset causes of the system.

Table 17-6. Members

Enum value	Description
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 BOD33.
SYSTEM_RESET_CAUSE_BOD12	The system was last reset by the BOD12.
SYSTEM_RESET_CAUSE_POR	The system was last reset by the POR (Power on reset).

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 17-7. Members

Enum value	Description
SYSTEM_SLEEPMODE_IDLE_0	IDLE 0 sleep mode.
SYSTEM_SLEEPMODE_IDLE_1	IDLE 1 sleep mode.
SYSTEM_SLEEPMODE_IDLE_2	IDLE 2 sleep mode.
SYSTEM_SLEEPMODE_STANDBY	Standby sleep mode.

Enum system_voltage_reference

List of available voltage references (VREF) that may be used within the device.

Table 17-8. Members

Enum value	Description
SYSTEM_VOLTAGE_REFERENCE_TEMPSENSE	Temperature sensor voltage reference.
SYSTEM_VOLTAGE_REFERENCE_BANDGAP	Bandgap voltage reference.

17.7 Extra Information for SYSTEM Driver

17.7.1 Acronyms

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

Acronym	Definition
PM	Power Manager
SYSCTRL	System control interface

17.7.2 Dependencies

This driver has the following dependencies:

- None

17.7.3 Errata

There are no errata related to this driver.

17.7.4 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

18. SAM D20 System Interrupt Driver

This driver for SAM D20 devices provides an interface for the configuration and management of internal software and hardware interrupts/exceptions.

The following peripherals are used by this module:

- NVIC (Nested Vector Interrupt Controller)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for System Interrupt
- Examples
- API Overview

18.1 Prerequisites

There are no prerequisites for this module.

18.2 Module Overview

The Cortex M0+ core contains an interrupt an 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.

18.2.1 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.

18.2.2 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.

18.3 Special Considerations

Interrupts from peripherals in the SAM D20 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.

18.4 Extra Information for System Interrupt

For extra information see Extra Information for SYSTEM INTERRUPT Driver. This includes:

- Acronyms
- Dependencies
- Errata

- Module History

18.5 Examples

For a list of examples related to this driver, see Examples for SYSTEM INTERRUPT Driver.

18.6 API Overview

18.6.1 Function Definitions

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.

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 18-1. 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 18-2. 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 18-3. 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 18-4. 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 18-5. Parameters

Data direction	Parameter name	Description
[in]	vector	Interrupt vector to disable

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 18-6. 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 18-7. 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 18-8. 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 18-9. 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 18-10. 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 18-11. Return Values

Return value	Description
STATUS_OK	If no error was detected
STATUS_INVALID_ARG	If an unsupported interrupt vector number was given

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 18-12. 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 18-13. 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 18-14. 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.

18.6.2 Enumeration Definitions

Enum system_interrupt_priority_level

Table of all possible interrupt and exception vector priorities within the device.

Table 18-15. 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.

Enum system_interrupt_vector

Table of all possible interrupt and exception vector indexes within the device.

Table 18-16. 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_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_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.

18.7 Extra Information for SYSTEM INTERRUPT Driver

18.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
ISR	Interrupt Service Routine

18.7.2 Dependencies

This driver has the following dependencies:

- None

18.7.3 Errata

There are no errata related to this driver.

18.7.4 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

18.8 Examples for SYSTEM INTERRUPT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 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 QSGs 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

18.8.1 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.

Setup

Prerequisites

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

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.

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.

```
system_interrupt_enter_critical_section();
```

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();
```

18.8.2 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.

Setup

Prerequisites

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

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();
```

19. SAM D20 Timer/Counter Driver (TC)

This driver for SAM D20 devices 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 peripherals are used by this module:

- TC (Timer/Counter)

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for TC
- Examples
- API Overview

19.1 Prerequisites

There are no prerequisites for this module.

19.2 Module Overview

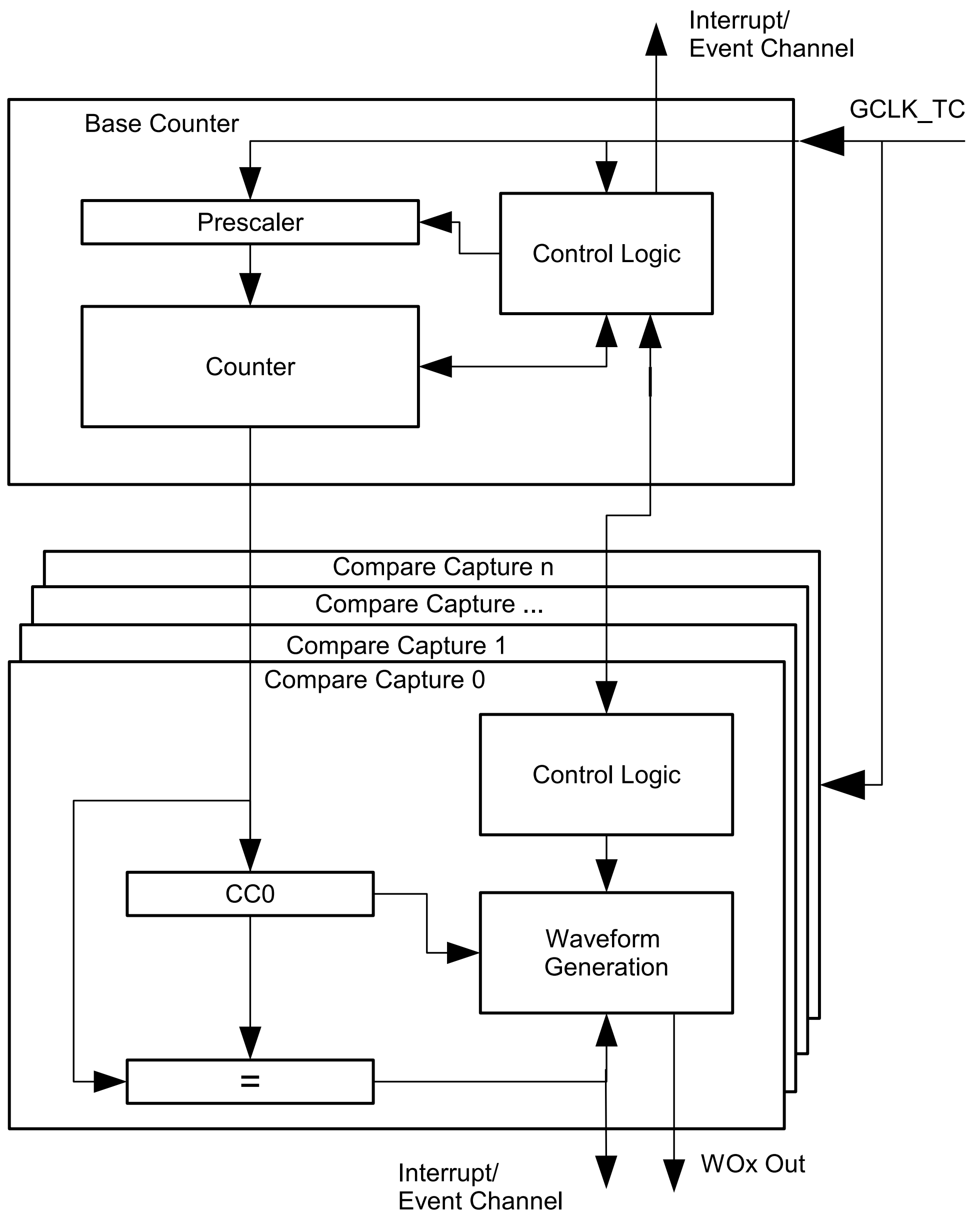
The Timer/Counter (TC) module provides a set of timing and counting related functionality, such as the generation of periodic waveforms, the capturing of a periodic waveform's frequency/duty cycle, and software timekeeping for periodic operations. TC modules can be configured to use an 8-, 16-, or 32-bit counter size.

This TC module for the SAM D20 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 19-1: Basic overview of the TC module shows the overview of the TC module design.

Figure 19-1. Basic overview of the TC module



19.2.1 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 PWM signal generation.

19.2.2 Timer/Counter Size

Each timer module can be configured in one of three different counter sizes; 8-, 16-, and 32-bits. 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 19-1: Timer counter sizes and their maximum count values shows the maximum values for each of the possible counter sizes.

Table 19-1. 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**
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. 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 19-2: TC master and slave module pairings.

Table 19-2. TC master and slave module pairings

Master TC Module	Slave TC Module
TC0	TC1
TC2	TC3
...	...
TCn-1	TCn

19.2.3 Clock Settings

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.

Prescaler

Each TC module in the SAM D20 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 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 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.

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 max 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 19-3: TC module reload actions.

Table 19-3. TC module reload actions

Reload Action	Description
TC_RELOAD_ACTION_GCLK [374]	Reload TC counter value on next GCLK cycle. Leave prescaler as-is.
TC_RELOAD_ACTION_PRESC [374]	Reloads TC counter value on next prescaler clock. Leave prescaler as-is.
TC_RELOAD_ACTION_RESYNC [374]	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 max 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.

19.2.4 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.

Basic Timer

A Basic Timer is a simple application where compare match operations 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 (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.

Waveform Generation

Waveform generation enables the TC module to generate square waves, or if combined with an external passive low-pass filter, analog waveforms.

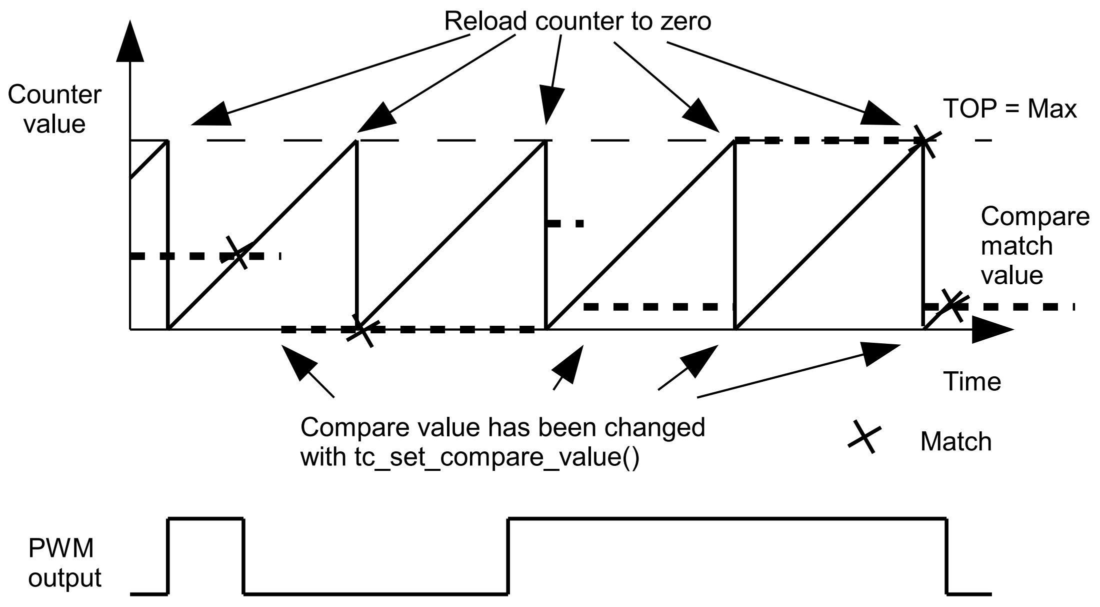
Waveform Generation - PWM

Pulse width modulation is a form of waveform generation and a signaling 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 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.

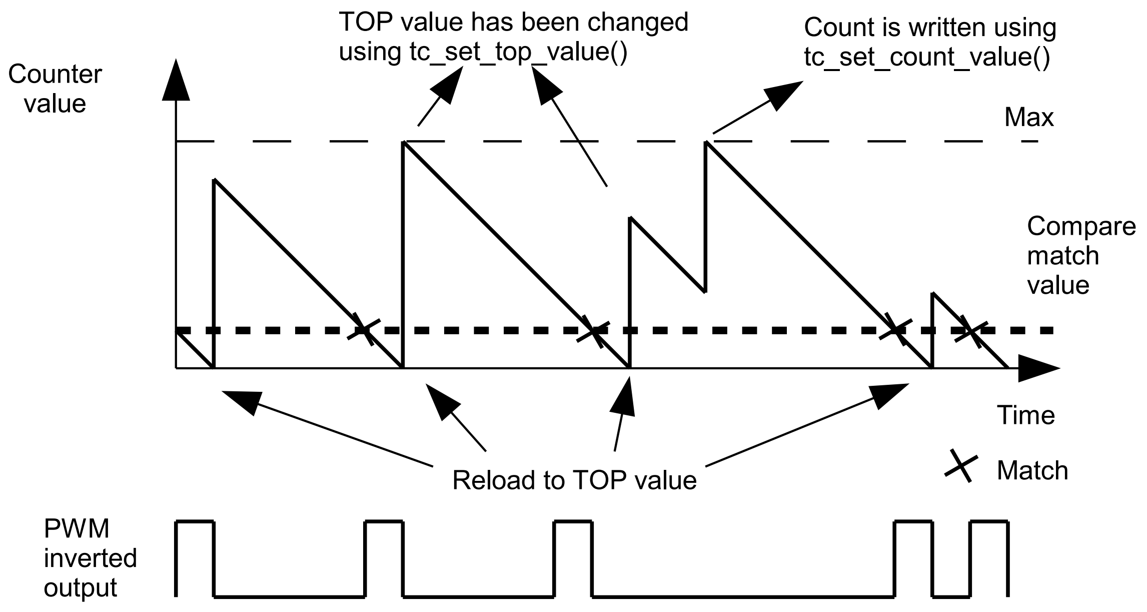
Figure 19-2: Example of PWM in normal mode, and different counter operations illustrates operations and different states of the counter and its output when running the counter in PWM normal mode. As can be seen, the TOP value is unchanged and is set to MAX. The compare match value is changed at several points to illustrate the resulting waveform output changes. The PWM output is set to normal (i.e. non-inverted) output mode.

Figure 19-2. Example of PWM in normal mode, and different counter operations



In Figure 19-3: Example of PWM in match mode, and different counter operations, 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 19-3. Example of PWM in match mode, and different counter operations


TOP value has been changed using tc_set_top_value()
Count is written using tc_set_count_value()
Counter value
Max
Compare match value
Time
Reload to TOP value
Match
PWM inverted output

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.

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.

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.

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 is 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.

19.2.5 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.

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.

19.3 Special Considerations

The number of capture compare registers in each TC module is dependent on the specific SAM D20 device being used, and in some cases the counter size.

The maximum amount of capture compare registers available in any SAMD20 device is two when running in 32-bit mode and four in 8-, and 16-bit modes.

19.4 Extra Information for TC

For extra information see Extra Information for TC Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

19.5 Examples

For a list of examples related to this driver, see Examples for TC Driver.

19.6 API Overview

19.6.1 Variable and Type Definitions

Type tc_callback_t

```
typedef void(* tc_callback_t )(struct tc_module *const module)
```

19.6.2 Structure Definitions

Struct tc_16bit_config

Table 19-4. Members

Type	Name	Description
uint16_t	compare_capture_channel[]	Value to be used for compare match on each channel.
uint16_t	count	Initial timer count value.

Struct tc_32bit_config

Table 19-5. Members

Type	Name	Description
uint32_t	compare_capture_channel[]	Value to be used for compare match on each channel.
uint32_t	count	Initial timer count value.

Struct tc_8bit_config

Table 19-6. Members

Type	Name	Description
uint8_t	compare_capture_channel[]	Value to be used for compare match on each channel.
uint8_t	count	Initial timer count value.
uint8_t	period	Where to count to or from depending on the direction on the counter.

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 19-7. Members

Type	Name	Description
bool	channel_pwm_out_enabled[]	When true, PWM output for the given channel is enabled.
uint32_t	channel_pwm_out_mux[]	Specifies MUX setting for each output channel pin.
uint32_t	channel_pwm_out_pin[]	Specifies pin output for each channel.
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	enable_capture_on_channel[]	Specifies which channel(s) to enable channel capture operation on.
enum tc_event_action	event_action	Specifies which event to trigger if an event is triggered.
bool	invert_event_input	Specifies if the input event source is inverted, when used in PWP or PPW event action modes.
bool	oneshot	When true, one-shot will stop the TC on next hardware or software re-trigger event or overflow/ underflow.
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 true the module is enabled during standby.
union tc_config.size_specific	size_specific	This setting determines what size counter is used.
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.

Union tc_config.size_specific

This setting determines what size counter is used.

Table 19-8. Members

Type	Name	Description
struct tc_16bit_config	size_16_bit	Struct for 16-bit specific timer configuration.

Type	Name	Description
struct tc_32bit_config	size_32_bit	Struct for 32-bit specific timer configuration.
struct tc_8bit_config	size_8_bit	Struct for 8-bit specific timer configuration.

Struct tc_events

Event flags for the tc_enable_events() and tc_disable_events().

Table 19-9. Members

Type	Name	Description
bool	generate_event_on_compare_chann	Generate an output event on a compare channel match.
bool	generate_event_on_overflow	Generate an output event on counter overflow.
bool	on_event_perform_action	Perform the configured event action when an incoming event is signaled.

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.

19.6.3 Macro Definitions

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 (1UL << 0)
```

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 (1UL << 1)
```

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 (1UL << 2)
```

Timer register synchronization has completed, and the synchronized count value may be read.

Macro TC_STATUS_CAPTURE_OVERFLOW

```
#define TC_STATUS_CAPTURE_OVERFLOW (1UL << 3)
```

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 (1UL << 4)
```

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.

19.6.4 Function Definitions

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 19-10. 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 19-11. Return Values

Return value	Description
true	if the module has completed synchronization
false	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
- No inversion of waveform output
- No capture enabled
- 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

Table 19-12. 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 19-13. Parameters

Data direction	Parameter name	Description
[inout]	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 19-14. 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

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 19-15. 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 19-16. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	events	Struct containing flags of events to disable

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 19-17. Parameters

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

Returns Status of the procedure

Table 19-18. Return Values

Return value	Description
STATUS_OK	The module was reset successfully

Return value	Description
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 19-19. 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 19-20. Parameters

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

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 19-21. 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 19-22. 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 19-23. Return Values

Return value	Description
STATUS_OK	The timer count was updated successfully
STATUS_ERR_INVALID_ARG	An invalid timer counter size was specified

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 max or the top value if the counter was counting down when stopped.

Table 19-24. 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 19-25. Parameters

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

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 19-26. 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 19-27. 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

Data direction	Parameter name	Description
[in]	compare	New compare value to set

Returns Status of the compare update procedure.

Table 19-28. Return Values

Return value	Description
STATUS_OK	The compare value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied

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 19-29. 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 19-30. 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.

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 19-31. Parameters

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

Returns Bitmask of TC_STATUS_* flags

Table 19-32. 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

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 19-33. 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

19.6.5 Enumeration Definitions

Enum tc_callback

Enum for the possible callback types for the TC module.

Table 19-34. 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

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 19-35. 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

Enum tc_compare_capture_channel

This enum is used to specify which capture/compare channel to do operations on.

Table 19-36. 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

Enum tc_count_direction

Timer/Counter count direction.

Table 19-37. 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.

Enum tc_counter_size

This enum specifies the maximum value it is possible to count to.

Table 19-38. Members

Enum value	Description
TC_COUNTER_SIZE_8BIT	The counter's max value is 0xFF, the period register is available to be used as top value.
TC_COUNTER_SIZE_16BIT	The counter's max 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 max 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.

Enum tc_event_action

Event action to perform when the module is triggered by an event.

Table 19-39. 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.

Enum tc_reload_action

This enum specify how the counter and prescaler should reload.

Table 19-40. 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.

Enum tc_wave_generation

This enum is used to select which mode to run the wave generation in.

Table 19-41. Members

Enum value	Description
TC_WAVE_GENERATION_NORMAL_FREQ	Top is max, except in 8-bit counter size where it is the PER register

Enum value	Description
TC_WAVE_GENERATION_MATCH_FREQ	Top is CC0, except in 8-bit counter size where it is the PER register
TC_WAVE_GENERATION_NORMAL_PWM	Top is max, 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

Enum tc_waveform_invert_output

Output waveform inversion mode.

Table 19-42. 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.

19.7 Extra Information for TC Driver

19.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
TC	Timer Counter
PWM	Pulse Width Modulation
PWP	Pulse Width Period
PPW	Period Pulse Width

19.7.2 Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

19.7.3 Errata

There are no errata related to this 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
Initial Release

19.8 Examples for TC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Timer/Counter Driver (TC). 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 - Callback

19.8.1 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

Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

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.size_specific.size_16_bit.compare_capture_channel[0] = (0xFFFF / 4);

    config_tc.channel_pwm_out_enabled[0] = true;
    config_tc.channel_pwm_out_pin[0]     = PWM_OUT_PIN;
    config_tc.channel_pwm_out_mux[0]     = 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.

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

```
struct tc_module tc_instance;
```

2. Configure the TC module.

   a. 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;
   ```

   b. Initialize the TC configuration struct with the module's default values.

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

   ```
   tc_get_config_defaults(&config_tc);
   ```

   c. 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.size_specific.size_16_bit.compare_capture_channel[0] = (0xFFFF / 4);
   ```

d. Alter the TC settings to configure the PWM output on a physical device pin.

```
config_tc.channel_pwm_out_enabled[0] = true;
config_tc.channel_pwm_out_pin[0]     = PWM_OUT_PIN;
config_tc.channel_pwm_out_mux[0]     = PWM_OUT_MUX;
```

e. Configure the TC module with the desired settings.

```
tc_init(&tc_instance, PWM_MODULE, &config_tc);
```

f. Enable the TC module to start the timer and begin PWM signal generation.

```
tc_enable(&tc_instance);
```

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 */
}
```

19.8.2 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

Quick Start

Prerequisites

There are no prerequisites for this use case.

Code

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.size_specific.size_16_bit.compare_capture_channel[0] = 0xFFFF;

    config_tc.channel_pwm_out_enabled[0] = true;
    config_tc.channel_pwm_out_pin[0]     = PWM_OUT_PIN;
    config_tc.channel_pwm_out_mux[0]     = 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.

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

```
struct tc_module tc_instance;
```

2. Configure the TC module.

   a. 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;
   ```

   b. Initialize the TC configuration struct with the module's default values.

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

   ```
   tc_get_config_defaults(&config_tc);
   ```

   c. 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.size_specific.size_16_bit.compare_capture_channel[0] = 0xFFFF;
   ```

   d. Alter the TC settings to configure the PWM output on a physical device pin.

   ```
   config_tc.channel_pwm_out_enabled[0] = true;
   config_tc.channel_pwm_out_pin[0]     = PWM_OUT_PIN;
   config_tc.channel_pwm_out_mux[0]     = PWM_OUT_MUX;
   ```

   e. Configure the TC module with the desired settings.

   ```
   tc_init(&tc_instance, PWM_MODULE, &config_tc);
   ```

   f. Enable the TC module to start the timer and begin PWM signal generation.

   ```
   tc_enable(&tc_instance);
   ```

3. Configure the TC callbacks.

a. 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);
```

b. 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);
```

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) {
}
```

20. SAM D20 Watchdog Driver (WDT)

This driver for SAM D20 devices 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 outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information for WDT
- Examples
- API Overview

20.1 Prerequisites

There are no prerequisites for this module.

20.2 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.

20.2.1 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.

20.2.2 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.

20.2.3 Early Warning

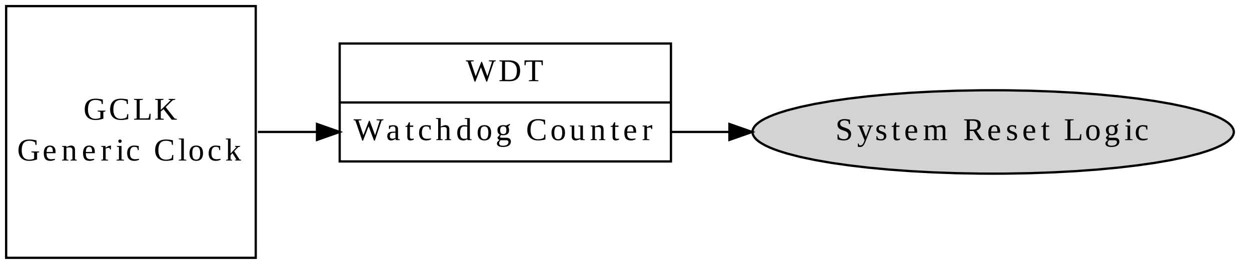
In some cases it is desirable to receive an early warning that the Watchdog is about to expire, so that some system action (such as saving any system configuration data for failure analysis purposes) can be performed before the system reset occurs. The Early Warning feature of the Watchdog module allows 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.

20.2.4 Physical Connection

Figure 20-1: Physical Connection shows how this module is interconnected within the device.

Figure 20-1. Physical Connection



20.3 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.

20.4 Extra Information for WDT

For extra information see Extra Information for WDT Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

20.5 Examples

For a list of examples related to this driver, see Examples for WDT Driver.

20.6 API Overview

20.6.1 Variable and Type Definitions

Callback configuration and initialization

Type wdt_callback_t

```
typedef void(* wdt_callback_t )(void)
```

Type definition for a WDT module callback function.

20.6.2 Structure Definitions

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 20-1. Members

Type	Name	Description
bool	always_on	If true, the Watchdog will be locked to the current configuration settings when the Watchdog is enabled.
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
enum wdt_period	early_warning_period	Number of Watchdog timer clock ticks until the early warning flag is set.
enum wdt_period	timeout_period	Number of Watchdog timer clock ticks until the Watchdog expires.
enum wdt_period	window_period	Number of Watchdog timer clock ticks until the reset window opens.

20.6.3 Function Definitions

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 20-2. 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 20-3. 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 20-4. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to unregister

Returns Status of the de-registration operation.

Table 20-5. Return Values

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

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 20-6. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to enable

Returns Status of the callback enable operation.

Table 20-7. 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 20-8. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to disable

Returns Status of the callback disable operation.

Table 20-9. Return Values

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

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 20-10. Return Values

Return value	Description
true	if the module has completed synchronization
false	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
- 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 20-11. Parameters

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

Function wdt_init()

Initializes and configures the Watchdog driver.

```
enum status_code wdt_init(
  const struct wdt_conf *const config)
```

Initializes the Watchdog driver, resetting the hardware module and configuring it to the user supplied configuration parameters, ready for use. This function should be called before enabling the Watchdog.

Note Once called the Watchdog will not be running; to start the Watchdog, call wdt_enable() after configuring the module.

Table 20-12. Parameters

Data direction	Parameter name	Description
[in]	config	Configuration settings for the Watchdog

Returns Status of the configuration procedure.

Table 20-13. 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_enable()

Enables the Watchdog Timer that was previously configured.

```
enum status_code wdt_enable(void)
```

Enables and starts the Watchdog Timer that was previously configured via a call to wdt_init().

Returns Status of the enable procedure.

Table 20-14. Return Values

Return value	Description
STATUS_OK	If the module was enabled correctly

Return value	Description
STATUS_ERR_IO	If the Watchdog module is locked to be always on

Function wdt_disable()

Disables the Watchdog Timer that was previously enabled.

```
enum status_code wdt_disable(void)
```

Stops the Watchdog Timer that was previously started via a call to wdt_enable().

Returns Status of the disable procedure.

Table 20-15. Return Values

Return value	Description
STATUS_OK	If the module was disabled correctly
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.

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.

20.6.4 Enumeration Definitions

Callback configuration and initialization

Enum wdt_callback

Enum for the possible callback types for the WDT module.

Table 20-16. Members

Enum value	Description
WDT_CALLBACK_EARLY_WARNING	Callback type for when an early warning callback from the WDT module is issued.

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 20-17. 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.

Enum value	Description
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.

20.7 Extra Information for WDT Driver

20.7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
WDT	Watchdog Timer

20.7.2 Dependencies

This driver has the following dependencies:

- System Clock Driver

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 WDT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM D20 Watchdog Driver (WDT). 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 WDT - Basic
- Quick Start Guide for WDT - Callback

20.8.1 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.

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;
    config_wdt.clock_source   = GCLK_GENERATOR_4;
    config_wdt.timeout_period = WDT_PERIOD_2048CLK;

    /* Initialize and enable the Watchdog with the user settings */
    wdt_init(&config_wdt);
    wdt_enable();
}
```

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.

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

```
wdt_get_config_defaults(&config_wdt);
```

3. Adjust the configuration struct to set the timeout period and lock mode of the Watchdog.

```
config_wdt.always_on      = false;
config_wdt.clock_source   = GCLK_GENERATOR_4;
config_wdt.timeout_period = WDT_PERIOD_2048CLK;
```

4. Initialize the Watchdog to configure the module with the requested settings.

```
wdt_init(&config_wdt);
```

5. Enable the Watchdog to start the module.

```
wdt_enable();
```

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();
```

20.8.2 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 boar LED is turned on, and each time the device resets the board LED is turned off, giving a periodic flashing pattern.

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;
    config_wdt.clock_source         = GCLK_GENERATOR_4;
    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_init(&config_wdt);
    wdt_enable();
}

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
    a. Create a Watchdog module configuration struct, which can be filled out to adjust the configuration of the Watchdog.

    ```
    struct wdt_conf config_wdt;
    ```

    b. Initialize the Watchdog configuration struct with the module's default values.

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

    ```
    wdt_get_config_defaults(&config_wdt);
    ```

    c. Adjust the configuration struct to set the timeout and early warning periods of the Watchdog.

    ```
    config_wdt.always_on            = false;
    config_wdt.clock_source         = GCLK_GENERATOR_4;
    config_wdt.timeout_period       = WDT_PERIOD_4096CLK;
    config_wdt.early_warning_period = WDT_PERIOD_2048CLK;
    ```

    d. Initialize the Watchdog to configure the module with the requested settings.

    ```
    wdt_init(&config_wdt);
    ```

    e. Enable the Watchdog to start the module.

    ```
    wdt_enable();
    ```

2. Register and enable the Early Warning callback handler
    a. 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);
    ```

    b. Enable the Early Warning callback so that it will generate callbacks.

    ```
    wdt_enable_callback(WDT_CALLBACK_EARLY_WARNING);
    ```

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 */
    }
    ```

21. Document Revision History

Doc. Rev.	Date	Comments
A	06/2013	Initial release