SMART ARM-based Microcontrollers

AT03249: SAM D/R/L/C RTC Count (RTC COUNT) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Real Time Clock functionality in Count operating mode, for the configuration and retrieval of the current RTC counter value. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- RTC (Real Time Clock)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

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

Table of Contents

1. Software License

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

The RTC module in the SAM devices is a 32-bit counter, with a 10-bit programmable prescaler. Typically, the RTC clock is run continuously, including in the device's low-power sleep modes, to track the current time and date information. The RTC can be used as a source to wake up the system at a scheduled time or periodically using the alarm functions.

In this driver, the RTC is operated in Count mode. This allows for an easy integration of an asynchronous counter into a user application, which is capable of operating while the device is in sleep mode.

Whilst operating in Count mode, the RTC features:

- 16-bit counter mode
  - Selectable counter period
  - Up to six configurable compare values
- 32-bit counter mode
  - Clear counter value on match
  - Up to four configurable compare values

3.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_RTC_PERIODIC_INT	SAM L21/L22/C20/C21
FEATURE_RTC_PRESCALER_OFF	SAM L21/L22/C20/C21
FEATURE_RTC_CLOCK_SELECTION	SAM L21/L22/C20/C21
FEATURE_RTC_GENERAL_PURPOSE_REG	SAM L21/L22
FEATURE_RTC_CONTINUOUSLY_UPDATED	SAM D20, SAM D21, SAM R21, SAM D10, SAM D11, SAM DA1
FEATURE_RTC_TAMPER_DETECTION	SAM L22

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

4. Compare and Overflow

The RTC can be used with up to 4/6 compare values (depending on selected operation mode). These compare values will trigger on match with the current RTC counter value, and can be set up to trigger an interrupt, event, or both. The RTC can also be configured to clear the counter value on compare match in 32-bit mode, resetting the count value back to zero.

If the RTC is operated without the Clear on Match option enabled, or in 16-bit mode, the RTC counter value will instead be cleared on overflow once the maximum count value has been reached:

$$COUNT_{MAX} = 2^{32} - 1$$

for 32-bit counter mode, and

$$COUNT_{MAX} = 2^{16} - 1$$

for 16-bit counter mode.

When running in 16-bit mode, the overflow value is selectable with a period value. The counter overflow will then occur when the counter value reaches the specified period value.

4.1. Periodic Events

The RTC can generate events at periodic intervals, allowing for direct peripheral actions without CPU intervention. The periodic events can be generated on the upper eight bits of the RTC prescaler, and will be generated on the rising edge transition of the specified bit. The resulting periodic frequency can be calculated by the following formula:

$$f_{PERIODIC} = \frac{f_{ASY}}{2^{n+3}}$$

Where

f_{ASY}

refers to the *asynchronous* clock is set up in the RTC module configuration. The **n** parameter is the event source generator index of the RTC module. If the asynchronous clock is operated at the recommended frequency of 1KHz, the formula results in the values shown in Table 4-1 RTC Event Frequencies for Each Prescaler Bit Using a 1KHz Clock on page 7.

Table 4-1 RTC Event Frequencies for Each Prescaler Bit Using a 1KHz Clock

n	Periodic event
7	1Hz
6	2Hz
5	4Hz
4	8Hz
3	16Hz
2	32Hz

n	Periodic event
1	64Hz
0	128Hz

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

4.2. Digital Frequency Correction

The RTC module contains Digital Frequency Correction logic to compensate for inaccurate source clock frequencies which would otherwise result in skewed time measurements. The correction scheme requires that at least two bits in the RTC module prescaler are reserved by the correction logic. As a result of this implementation, frequency correction is only available when the RTC is running from a 1Hz reference clock.

The correction procedure is implemented by subtracting or adding a single cycle from the RTC prescaler every 1024 RTC GCLK cycles. The adjustment is applied the specified number of time (maximum 127) over 976 of these periods. The corresponding correction in PPM will be given by:

$$Correction(PPM) = \frac{VALUE}{999424}10^6$$

The RTC clock will tick faster if provided with a positive correction value, and slower when given a negative correction value.

4.3. RTC Tamper Detect

see RTC Tamper Detect

5. Special Considerations

5.1. Clock Setup

5.1.1. SAM D20/D21/R21/D10/D11/DA1 Clock Setup

The RTC is typically clocked by a specialized GCLK generator that has a smaller prescaler than the others. By default the RTC clock is on, selected to use the internal 32KHz RC-oscillator with a prescaler of 32, giving a resulting clock frequency of 1KHz to the RTC. When the internal RTC prescaler is set to 1024, this yields an end-frequency of 1Hz.

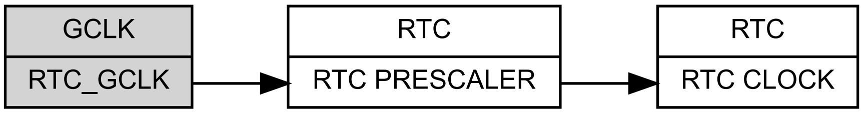
The implementer also has the option to set other end-frequencies. Table 5-1 RTC Output Frequencies from Allowable Input Clocks on page 9 lists the available RTC frequencies for each possible GCLK and RTC input prescaler options.

Table 5-1 RTC Output Frequencies from Allowable Input Clocks

End-frequency	GCLK prescaler	RTC prescaler
32KHz	1	1
1KHz	32	1
1Hz	32	1024

The overall RTC module clocking scheme is shown in Figure 5-1 SAM D20/D21/R21/D10/D11/DA1 Clock Setup on page 9.

Figure 5-1 SAM D20/D21/R21/D10/D11/DA1 Clock Setup



5.1.2. SAM L21/C20/C21 Clock Setup

The RTC clock can be selected from OSC32K, XOSC32K, or OSCULP32K, and a 32KHz or 1KHz oscillator clock frequency is required. This clock must be configured and enabled in the 32KHz oscillator controller before using the RTC.

The table below lists the available RTC clock Table 5-2 RTC Clocks Source on page 9.

Table 5-2 RTC Clocks Source

RTC clock frequency	Clock source	Description
1.024KHz	ULP1K	1.024KHz from 32KHz internal ULP oscillator
32.768KHz	ULP32K	32.768KHz from 32KHz internal ULP oscillator
1.024KHz	OSC1K	1.024KHz from 32KHz internal oscillator
32.768KHz	OSC32K	32.768KHz from 32KHz internal oscillator

RTC clock frequency	Clock source	Description
1.024KHz	XOSC1K	1.024KHz from 32KHz internal oscillator
32.768KHz	XOSC32K	32.768KHz from 32KHz external crystal oscillator

6. Extra Information

For extra information, see Extra Information for RTC COUNT Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

7. Examples

For a list of examples related to this driver, see Examples for RTC (COUNT) Driver.

8. API Overview

8.1. Structure Definitions

8.1.1. Struct rtc_count_config

Configuration structure for the RTC instance. This structure should be initialized using the rtc_count_get_config_defaults() before any user configurations are set.

Table 8-1 Members

Type	Name	Description
bool	clear_on_match	If true, clears the counter value on compare match. Only available whilst running in 32-bit mode
uint32_t	compare_values[]	Array of Compare values. Not all Compare values are available in 32-bit mode
enum rtc_count_mode	mode	Select the operation mode of the RTC
enum rtc_count_prescaler	prescaler	Input clock prescaler for the RTC module

8.1.2. Struct rtc_count_events

Event flags for the rtc_count_enable_events() and rtc_count_disable_events().

Table 8-2 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
bool	generate_event_on_tamper	Generate an output event on every tamper input
bool	on_event_to_tamper	Tamper input event and capture the COUNT value

8.1.3. Struct rtc_tamper_config

The configuration structure for the RTC tamper. This structure should be initialized using the rtc_tamper_get_config_defaults() before any user configurations are set.

Table 8-3 Members

Type	Name	Description
enum rtc_tamper_active_layer_freq_divider	actl_freq_div	Active layer frequency
bool	bkup_reset_on_tamper	Backup register reset on tamper enable
enum rtc_tamper_debounce_freq_divider	deb_freq_div	Debounce frequency
enum rtc_tamper_debounce_seq	deb_seq	Debounce sequential
bool	dma_tamper_enable	DMA on tamper enable
bool	gp0_enable	General Purpose 0/1 Enable
bool	gp_reset_on_tamper	GP register reset on tamper enable
struct rtc_tamper_input_config	in_cfg[]	Tamper IN configuration

8.1.4. Struct rtc_tamper_input_config

The configuration structure for tamper INn.

Table 8-4 Members

Type	Name	Description
enum rtc_tamper_input_action	action	Tamper input action
bool	debounce_enable	Debounce enable
enum rtc_tamper_level_sel	level	Tamper level select

8.2. Macro Definitions

8.2.1. Driver Feature Definition

Define port features set according to different device family.

8.2.1.1. Macro FEATURE_RTC_PERIODIC_INT

```
#define FEATURE_RTC_PERIODIC_INT
```

RTC periodic interval interrupt.

8.2.1.2. Macro FEATURE_RTC_PRESCALER_OFF

```
#define FEATURE_RTC_PRESCALER_OFF
```

RTC prescaler is off.

8.2.1.3. Macro FEATURE_RTC_CLOCK_SELECTION

```
#define FEATURE_RTC_CLOCK_SELECTION
```

RTC clock selection.

8.2.1.4. Macro FEATURE_RTC_GENERAL_PURPOSE_REG

```
#define FEATURE_RTC_GENERAL_PURPOSE_REG
```

General purpose registers.

8.2.1.5. Macro FEATURE_RTC_TAMPER_DETECTION

```
#define FEATURE_RTC_TAMPER_DETECTION
```

RTC tamper detection.

8.2.2. Macro RTC_TAMPER_DETECT_EVT

```
#define RTC_TAMPER_DETECT_EVT
```

RTC tamper input event detection bitmask.

8.2.3. Macro RTC_TAMPER_DETECT_ID0

```
#define RTC_TAMPER_DETECT_ID0
```

RTC tamper ID0 detection bitmask.

8.2.4. Macro RTC_TAMPER_DETECT_ID1

```
#define RTC_TAMPER_DETECT_ID1
```

RTC tamper ID1 detection bitmask.

8.2.5. Macro RTC_TAMPER_DETECT_ID2

```
#define RTC_TAMPER_DETECT_ID2
```

RTC tamper ID2 detection bitmask.

8.2.6. Macro RTC_TAMPER_DETECT_ID3

```
#define RTC_TAMPER_DETECT_ID3
```

RTC tamper ID3 detection bitmask.

8.2.7. Macro RTC_TAMPER_DETECT_ID4

```
#define RTC_TAMPER_DETECT_ID4
```

RTC tamper ID4 detection bitmask.

8.3. Function Definitions

8.3.1. Configuration and Initialization

8.3.1.1. Function rtc_count_get_config_defaults()

Gets the RTC default configurations.

```
void rtc_count_get_config_defaults(
        struct rtc_count_config *const config)
```

Initializes the configuration structure to default values. This function should be called at the start of any RTC initialization.

The default configuration is:
- Input clock divided by a factor of 1024
- RTC in 32-bit mode
- Clear on compare match off
- Continuously sync count register off
- No event source on
- All compare values equal 0
- Count read synchronization is enabled for SAM L22

Table 8-5 Parameters

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

8.3.1.2. Function rtc_count_reset()

Resets the RTC module. Resets the RTC to hardware defaults.

```
void rtc_count_reset(
        struct rtc_module *const module)
```

Table 8-6 Parameters

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

8.3.1.3. Function rtc_count_enable()

Enables the RTC module.

```
void rtc_count_enable(
        struct rtc_module *const module)
```

Enables the RTC module once it has been configured, ready for use. Most module configuration parameters cannot be altered while the module is enabled.

Table 8-7 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

8.3.1.4. Function rtc_count_disable()

Disables the RTC module.

```
void rtc_count_disable(
        struct rtc_module *const module)
```

Disables the RTC module.

Table 8-8 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

8.3.1.5. Function rtc_count_init()

Initializes the RTC module with given configurations.

```
enum status_code rtc_count_init(
        struct rtc_module *const module,
        Rtc *const hw,
        const struct rtc_count_config *const config)
```

Initializes the module, setting up all given configurations to provide the desired functionality of the RTC.

Table 8-9 Parameters

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

Returns
Status of the initialization procedure.

Table 8-10 Return Values

Return value	Description
STATUS_OK	If the initialization was run stressfully
STATUS_ERR_INVALID_ARG	If invalid argument(s) were given

8.3.1.6. Function rtc_count_frequency_correction()

Calibrate for too-slow or too-fast oscillator.

```
enum status_code rtc_count_frequency_correction(
        struct rtc_module *const module,
        const int8_t value)
```

When used, the RTC will compensate for an inaccurate oscillator. The RTC module will add or subtract cycles from the RTC prescaler to adjust the frequency in approximately 1 PPM steps. The provided correction value should be between 0 and 127, allowing for a maximum 127 PPM correction.

If no correction is needed, set value to zero.

Note: Can only be used when the RTC is operated in 1Hz.

Table 8-11 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	value	Ranging from -127 to 127 used for the correction

Returns
Status of the calibration procedure.

Table 8-12 Return Values

Return value	Description
STATUS_OK	If calibration was executed correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

8.3.2. Count and Compare Value Management

8.3.2.1. Function rtc_count_set_count()

Set the current count value to desired value.

```
enum status_code rtc_count_set_count(
        struct rtc_module *const module,
        const uint32_t count_value)
```

Sets the value of the counter to the specified value.

Table 8-13 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	count_value	The value to be set in count register

Returns
Status of setting the register.

Table 8-14 Return Values

Return value	Description
STATUS_OK	If everything was executed correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

8.3.2.2. Function rtc_count_get_count()

Get the current count value.

```
uint32_t rtc_count_get_count(
        struct rtc_module *const module)
```

Table 8-15 Parameters

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

Returns the current count value.

Returns
The current counter value as a 32-bit unsigned integer.

8.3.2.3. Function rtc_count_set_compare()

Set the compare value for the specified compare.

```
enum status_code rtc_count_set_compare(
        struct rtc_module *const module,
        const uint32_t comp_value,
        const enum rtc_count_compare comp_index)
```

Sets the value specified by the implementer to the requested compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 8-16 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	comp_value	The value to be written to the compare
[in]	comp_index	Index of the compare to set

Returns
Status indicating if compare was successfully set.

Table 8-17 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

8.3.2.4. Function rtc_count_get_compare()

Get the current compare value of specified compare.

```
enum status_code rtc_count_get_compare(
        struct rtc_module *const module,
        uint32_t *const comp_value,
        const enum rtc_count_compare comp_index)
```

Retrieves the current value of the specified compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 8-18 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	comp_value	Pointer to 32-bit integer that will be populated with the current compare value
[in]	comp_index	Index of compare to check

Returns
Status of the reading procedure.

Table 8-19 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

8.3.2.5. Function rtc_count_set_period()

Set the given value to the period.

```
enum status_code rtc_count_set_period(
        struct rtc_module *const module,
        uint16_t period_value)
```

Sets the given value to the period.

Note: Only available in 16-bit mode.

Table 8-20 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	period_value	The value to set to the period

Returns
Status of setting the period value.

Table 8-21 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

8.3.2.6. Function rtc_count_get_period()

Retrieves the value of period.

```
enum status_code rtc_count_get_period(
        struct rtc_module *const module,
        uint16_t *const period_value)
```

Retrieves the value of the period for the 16-bit mode counter.

Note: Only available in 16-bit mode.

Table 8-22 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[out]	period_value	Pointer to value for return argument

Returns
Status of getting the period value.

Table 8-23 Return Values

Return value	Description
STATUS_OK	If the period value was read correctly
STATUS_ERR_UNSUPPORTED_DEV	If incorrect mode was set

8.3.3. Status Management

8.3.3.1. Function rtc_count_is_overflow()

Check if an RTC overflow has occurred.

```
bool rtc_count_is_overflow(
        struct rtc_module *const module)
```

Checks the overflow flag in the RTC. The flag is set when there is an overflow in the clock.

Table 8-24 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

Returns
Overflow state of the RTC module.

Table 8-25 Return Values

Return value	Description
true	If the RTC count value has overflowed
false	If the RTC count value has not overflowed

8.3.3.2. Function rtc_count_clear_overflow()

Clears the RTC overflow flag.

```
void rtc_count_clear_overflow(
        struct rtc_module *const module)
```

Clears the RTC module counter overflow flag, so that new overflow conditions can be detected.

Table 8-26 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module

8.3.3.3. Function rtc_count_is_periodic_interval()

Check if an RTC periodic interval interrupt has occurred.

```
bool rtc_count_is_periodic_interval(
        struct rtc_module *const module,
        enum rtc_count_periodic_interval n)
```

Checks the periodic interval flag in the RTC.

Table 8-27 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	n	RTC periodic interval interrupt

Returns
Periodic interval interrupt state of the RTC module.

Table 8-28 Return Values

Return value	Description
true	RTC periodic interval interrupt occurs
false	RTC periodic interval interrupt doesn't occur

8.3.3.4. Function rtc_count_clear_periodic_interval()

Clears the RTC periodic interval flag.

```
void rtc_count_clear_periodic_interval(
        struct rtc_module *const module,
        enum rtc_count_periodic_interval n)
```

Clears the RTC module counter periodic interval flag, so that new periodic interval conditions can be detected.

Table 8-29 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	n	RTC periodic interval interrupt

8.3.3.5. Function rtc_count_is_compare_match()

Check if RTC compare match has occurred.

```
bool rtc_count_is_compare_match(
        struct rtc_module *const module,
        const enum rtc_count_compare comp_index)
```

Checks the compare flag to see if a match has occurred. The compare flag is set when there is a compare match between counter and the compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 8-30 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	comp_index	Index of compare to check current flag

8.3.3.6. Function rtc_count_clear_compare_match()

Clears RTC compare match flag.

```
enum status_code rtc_count_clear_compare_match(
        struct rtc_module *const module,
        const enum rtc_count_compare comp_index)
```

Clears the compare flag. The compare flag is set when there is a compare match between the counter and the compare.

Note: Compare 4 and 5 are only available in 16-bit mode.

Table 8-31 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	comp_index	Index of compare to check current flag

Returns
Status indicating if flag was successfully cleared.

Table 8-32 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

8.3.4. Event Management

8.3.4.1. Function rtc_count_enable_events()

Enables an RTC event output.

```
void rtc_count_enable_events(
        struct rtc_module *const module,
        struct rtc_count_events *const events)
```

Enables one or more output events from the RTC module. See rtc_count_events for a list of events this module supports.

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

Table 8-33 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	events	Struct containing flags of events to enable

8.3.4.2. Function rtc_count_disable_events()

Disables an RTC event output.

```
void rtc_count_disable_events(
        struct rtc_module *const module,
        struct rtc_count_events *const events)
```

Disabled one or more output events from the RTC module. See rtc_count_events for a list of events this module supports.

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

Table 8-34 Parameters

Data direction	Parameter name	Description
[in, out]	module	RTC hardware module
[in]	events	Struct containing flags of events to disable

8.3.5. RTC General Purpose Registers

8.3.5.1. Function rtc_write_general_purpose_reg()

Write a value into general purpose register.

```
void rtc_write_general_purpose_reg(
        struct rtc_module *const module,
        const uint8_t index,
        uint32_t value)
```

Table 8-35 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	n	General purpose type
[in]	index	General purpose register index (0..3)

8.3.5.2. Function rtc_read_general_purpose_reg()

Read the value from general purpose register.

```
uint32_t rtc_read_general_purpose_reg(
        struct rtc_module *const module,
        const uint8_t index)
```

Table 8-36 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	index	General purpose register index (0..3)

Returns
Value of general purpose register.

8.3.6. Callbacks

8.3.6.1. Function rtc_count_register_callback()

Registers callback for the specified callback type.

```
enum status_code rtc_count_register_callback(
        struct rtc_module *const module,
        rtc_count_callback_t callback,
        enum rtc_count_callback callback_type)
```

Associates the given callback function with the specified callback type. To enable the callback, the rtc_count_enable_callback function must be used.

Table 8-37 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback	Pointer to the function desired for the specified callback
[in]	callback_type	Callback type to register

Returns
Status of registering callback.

Table 8-38 Return Values

Return value	Description
STATUS_OK	Registering was done successfully
STATUS_ERR_INVALID_ARG	If trying to register a callback not available

8.3.6.2. Function rtc_count_unregister_callback()

Unregisters callback for the specified callback type.

```
enum status_code rtc_count_unregister_callback(
        struct rtc_module *const module,
        enum rtc_count_callback callback_type)
```

When called, the currently registered callback for the given callback type will be removed.

Table 8-39 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	callback_type	Specifies the callback type to unregister

Returns
Status of unregistering callback.

Table 8-40 Return Values

Return value	Description
STATUS_OK	Unregistering was done successfully
STATUS_ERR_INVALID_ARG	If trying to unregister a callback not available

8.3.6.3. Function rtc_count_enable_callback()

Enables callback.

```
void rtc_count_enable_callback(
        struct rtc_module *const module,
        enum rtc_count_callback callback_type)
```

Enables the callback specified by the callback_type.

Table 8-41 Parameters

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

8.3.6.4. Function rtc_count_disable_callback()

Disables callback.

```
void rtc_count_disable_callback(
        struct rtc_module *const module,
        enum rtc_count_callback callback_type)
```

Disables the callback specified by the callback_type.

Table 8-42 Parameters

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

8.3.7. RTC Tamper Detection

8.3.7.1. Function rtc_tamper_get_config_defaults()

Gets the RTC tamper default configurations.

```
void rtc_tamper_get_config_defaults(
        struct rtc_tamper_config *const config)
```

Initializes the configuration structure to default values.

The default configuration is as follows:

- Disable backup register reset on tamper
- Disable GP register reset on tamper
- Active layer clock divided by a factor of 8
- Debounce clock divided by a factor of 8
- Detect edge on INn with synchronous stability debouncing
- Disable DMA on tamper
- Enable GP register
- Disable debouce, detect on falling edge and no action on INn

Table 8-43 Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to be initialized to default values.

8.3.7.2. Function rtc_tamper_set_config()

```
enum status_code rtc_tamper_set_config(
        struct rtc_module *const module,
        struct rtc_tamper_config *const tamper_cfg)
```

8.3.7.3. Function rtc_tamper_get_detect_flag()

Retrieves the RTC tamper detection status.

```
uint32_t rtc_tamper_get_detect_flag(
        struct rtc_module *const module)
```

Retrieves the detection status of each input pin and the input event.

Table 8-44 Parameters

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

Returns
Bitmask of detection flags.

Table 8-45 Return Values

Return value	Description
RTC_TAMPER_DETECT_ID0	Tamper condition on IN0 has been detected
RTC_TAMPER_DETECT_ID1	Tamper condition on IN1 has been detected
RTC_TAMPER_DETECT_ID2	Tamper condition on IN2 has been detected
RTC_TAMPER_DETECT_ID3	Tamper condition on IN3 has been detected
RTC_TAMPER_DETECT_ID4	Tamper condition on IN4 has been detected
RTC_TAMPER_DETECT_EVT	Tamper input event has been detected

8.3.7.4. Function rtc_tamper_clear_detect_flag()

Clears RTC tamper detection flag.

```
void rtc_tamper_clear_detect_flag(
        struct rtc_module *const module,
        const uint32_t detect_flags)
```

Clears the given detection flag of the module.

Table 8-46 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the TC software instance struct
[in]	detect_flags	Bitmask of detection flags

8.3.8. Function rtc_tamper_get_stamp()

Get the tamper stamp value.

```
uint32_t rtc_tamper_get_stamp(
        struct rtc_module *const module)
```

Table 8-47 Parameters

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

Returns
The current tamper stamp value as a 32-bit unsigned integer.

8.4. Enumeration Definitions

8.4.1. Enum rtc_clock_sel

Table 8-48 Members

Enum value	Description
RTC_CLOCK_SELECTION_ULP1K	1.024KHz from 32KHz internal ULP oscillator
RTC_CLOCK_SELECTION_ULP32K	32.768KHz from 32KHz internal ULP oscillator
RTC_CLOCK_SELECTION_OSC1K	1.024KHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_OSC32K	32.768KHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_XOSC1K	1.024KHz from 32KHz internal oscillator
RTC_CLOCK_SELECTION_XOSC32K	32.768KHz from 32.768KHz external crystal oscillator

8.4.2. Enum rtc_count_callback

The available callback types for the RTC count module.

Table 8-49 Members

Enum value	Description
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_0	Callback for Periodic Interval 0 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_1	Callback for Periodic Interval 1 Interrupt

Enum value	Description
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_2	Callback for Periodic Interval 2 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_3	Callback for Periodic Interval 3 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_4	Callback for Periodic Interval 4 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_5	Callback for Periodic Interval 5 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_6	Callback for Periodic Interval 6 Interrupt
RTC_COUNT_CALLBACK_PERIODIC_INTERVAL_7	Callback for Periodic Interval 7 Interrupt
RTC_COUNT_CALLBACK_COMPARE_0	Callback for compare channel 0
RTC_COUNT_CALLBACK_COMPARE_1	Callback for compare channel 1
RTC_COUNT_CALLBACK_COMPARE_2	Callback for compare channel 2
RTC_COUNT_CALLBACK_COMPARE_3	Callback for compare channel 3
RTC_COUNT_CALLBACK_COMPARE_4	Callback for compare channel 4
RTC_COUNT_CALLBACK_COMPARE_5	Callback for compare channel 5
RTC_COUNT_CALLBACK_TAMPER	Callback for tamper
RTC_COUNT_CALLBACK_OVERFLOW	Callback for overflow

8.4.3. Enum rtc_count_compare

Note: Not all compare channels are available in all devices and modes.

Table 8-50 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

8.4.4. Enum rtc_count_mode

RTC Count operating modes, to select the counting width and associated module operation.

Table 8-51 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

8.4.5. Enum rtc_count_periodic_interval

Table 8-52 Members

Enum value	Description
RTC_COUNT_PERIODIC_INTERVAL_0	Periodic interval 0
RTC_COUNT_PERIODIC_INTERVAL_1	Periodic interval 1
RTC_COUNT_PERIODIC_INTERVAL_2	Periodic interval 2
RTC_COUNT_PERIODIC_INTERVAL_3	Periodic interval 3
RTC_COUNT_PERIODIC_INTERVAL_4	Periodic interval 4
RTC_COUNT_PERIODIC_INTERVAL_5	Periodic interval 5
RTC_COUNT_PERIODIC_INTERVAL_6	Periodic interval 6
RTC_COUNT_PERIODIC_INTERVAL_7	Periodic interval 7

8.4.6. Enum rtc_count_prescaler

The available input clock prescaler values for the RTC count module.

Table 8-53 Members

Enum value	Description
RTC_COUNT_PRESCALER_OFF	RTC prescaler is off, and the input clock frequency is prescaled by a factor of 1
RTC_COUNT_PRESCALER_DIV_1	RTC input clock frequency is prescaled by a factor of 1
RTC_COUNT_PRESCALER_DIV_2	RTC input clock frequency is prescaled by a factor of 2
RTC_COUNT_PRESCALER_DIV_4	RTC input clock frequency is prescaled by a factor of 4
RTC_COUNT_PRESCALER_DIV_8	RTC input clock frequency is prescaled by a factor of 8
RTC_COUNT_PRESCALER_DIV_16	RTC input clock frequency is prescaled by a factor of 16
RTC_COUNT_PRESCALER_DIV_32	RTC input clock frequency is prescaled by a factor of 32
RTC_COUNT_PRESCALER_DIV_64	RTC input clock frequency is prescaled by a factor of 64
RTC_COUNT_PRESCALER_DIV_128	RTC input clock frequency is prescaled by a factor of 128
RTC_COUNT_PRESCALER_DIV_256	RTC input clock frequency is prescaled by a factor of 256

Enum value	Description
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

8.4.7. Enum rtc_tamper_active_layer_freq_divider

The available prescaler factor for the RTC clock output used during active layer protection.

Table 8-54 Members

Enum value	Description
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_2	RTC active layer frequency is prescaled by a factor of 2
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_4	RTC active layer frequency is prescaled by a factor of 4
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_8	RTC active layer frequency is prescaled by a factor of 8
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_16	RTC active layer frequency is prescaled by a factor of 16
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_32	RTC active layer frequency is prescaled by a factor of 32
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_64	RTC active layer frequency is prescaled by a factor of 64
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_128	RTC active layer frequency is prescaled by a factor of 128
RTC_TAMPER_ACTIVE_LAYER_FREQ_DIV_256	RTC active layer frequency is prescaled by a factor of 256

8.4.8. Enum rtc_tamper_debounce_freq_divider

The available prescaler factor for the input debouncers.

Table 8-55 Members

Enum value	Description
RTC_TAMPER_DEBOUNCE_FREQ_DIV_2	RTC debounce frequency is prescaled by a factor of 2
RTC_TAMPER_DEBOUNCE_FREQ_DIV_4	RTC debounce frequency is prescaled by a factor of 4
RTC_TAMPER_DEBOUNCE_FREQ_DIV_8	RTC debounce frequency is prescaled by a factor of 8
RTC_TAMPER_DEBOUNCE_FREQ_DIV_16	RTC debounce frequency is prescaled by a factor of 16
RTC_TAMPER_DEBOUNCE_FREQ_DIV_32	RTC debounce frequency is prescaled by a factor of 32
RTC_TAMPER_DEBOUNCE_FREQ_DIV_64	RTC debounce frequency is prescaled by a factor of 64

Enum value	Description
RTC_TAMPER_DEBOUNCE_FREQ_DIV_128	RTC debounce frequency is prescaled by a factor of 128
RTC_TAMPER_DEBOUNCE_FREQ_DIV_256	RTC debounce frequency is prescaled by a factor of 256

8.4.9. Enum rtc_tamper_debounce_seq

The available sequential for tamper debounce.

Table 8-56 Members

Enum value	Description
RTC_TAMPER_DEBOUNCE_SYNC	Tamper input detect edge with synchronous stability debounce
RTC_TAMPER_DEBOUNCE_ASYNC	Tamper input detect edge with asynchronous stability debounce
RTC_TAMPER_DEBOUNCE_MAJORITY	Tamper input detect edge with majority debounce

8.4.10. Enum rtc_tamper_input_action

The available action taken by the tamper input.

Table 8-57 Members

Enum value	Description
RTC_TAMPER_INPUT_ACTION_OFF	RTC tamper input action is disabled
RTC_TAMPER_INPUT_ACTION_WAKE	RTC tamper input action is wake and set tamper flag
RTC_TAMPER_INPUT_ACTION_CAPTURE	RTC tamper input action is capture timestamp and set tamper flag
RTC_TAMPER_INPUT_ACTION_ACTL	RTC tamper input action is compare IN to OUT, when a mismatch occurs, capture timestamp and set tamper flag

8.4.11. Enum rtc_tamper_level_sel

The available edge condition for tamper INn level select.

Table 8-58 Members

Enum value	Description
RTC_TAMPER_LEVEL_FALLING	A falling edge condition will be detected on Tamper input
RTC_TAMPER_LEVEL_RISING	A rising edge condition will be detected on Tamper input

9. RTC Tamper Detect

The RTC provides several selectable polarity external inputs (INn) that can be used for tamper detection. When detect, tamper inputs support the four actions:

- Off
- Wake
- Capture
- Active layer protection

Note: The Active Layer Protection is a means of detecting broken traces on the PCB provided by RTC. In this mode an RTC output signal is routed over critical components onthe board and fed back to one of the RTC inputs. The input and output signals are compared and a tamper condition is detected when they do not match.

Separate debouncers are embedded for each external input. The detection time depends on whether the debouncer operates synchronously or asynchronously, and whether majority detection is enabled or not. Details refer to the section "Tamper Detection" of datasheet.

10. Extra Information for RTC COUNT Driver

10.1. 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

10.2. Dependencies

This driver has the following dependencies:

- None

10.3. Errata

There are no errata related to this driver.

10.4. Module History

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

Changelog
Added support for SAM C21
Added support for SAM L21/L22
Added support for RTC tamper feature
Added driver instance parameter to all API function calls, except get_config_defaults
Updated initialization function to also enable the digital interface clock to the module if it is disabled
Initial Release

11. Examples for RTC (COUNT) Driver

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

- Quick Start Guide for RTC (COUNT) - Basic
- Quick Start Guide for RTC (COUNT) - Callback
- Quick Start Guide for RTC Tamper with DMA

11.1. Quick Start Guide for RTC (COUNT) - Basic

In this use case, the RTC is set up in count mode. The example configures the RTC in 16-bit mode, with continuous updates to the COUNT register, together with a set compare register value. Every 2000ms a LED on the board is toggled.

11.1.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

11.1.1.1. Clocks and Oscillators

The `conf_clock.h` file needs to be changed with the following values to configure the clocks and oscillators for the module.

The following oscillator settings are needed:

```
/* SYSTEM_CLOCK_SOURCE_OSC32K configuration - Internal 32KHz oscillator */
#  define CONF_CLOCK_OSC32K_ENABLE                true
#  define CONF_CLOCK_OSC32K_STARTUP_TIME          SYSTEM_OSC32K_STARTUP_130
#  define CONF_CLOCK_OSC32K_ENABLE_1KHZ_OUTPUT    true
#  define CONF_CLOCK_OSC32K_ENABLE_32KHZ_OUTPUT   true
#  define CONF_CLOCK_OSC32K_ON_DEMAND             true
#  define CONF_CLOCK_OSC32K_RUN_IN_STANDBY        false
```

The following generic clock settings are needed:

```
/* Configure GCLK generator 2 (RTC) */
#  define CONF_CLOCK_GCLK_2_ENABLE                true
#  define CONF_CLOCK_GCLK_2_RUN_IN_STANDBY        false
#  define CONF_CLOCK_GCLK_2_CLOCK_SOURCE
SYSTEM_CLOCK_SOURCE_OSC32K
#  define CONF_CLOCK_GCLK_2_PRESCALER             32
#  define CONF_CLOCK_GCLK_2_OUTPUT_ENABLE         false
```

11.1.2. Setup

11.1.2.1. Initialization Code

Create an rtc_module struct and add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

Copy-paste the following setup code to your applications `main()`:

```
void configure_rtc_count(void)
{
    struct rtc_count_config config_rtc_count;

    rtc_count_get_config_defaults(&config_rtc_count);

    config_rtc_count.prescaler            = RTC_COUNT_PRESCALER_DIV_1;
    config_rtc_count.mode                 = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif
    config_rtc_count.compare_values[0]   = 1000;
    rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

    rtc_count_enable(&rtc_instance);
}
```

11.1.2.2. Add to Main

Add the following to your `main()`.

```
configure_rtc_count();
```

11.1.2.3. Workflow

1. Create an RTC configuration structure to hold the desired RTC driver settings.

   ```
   struct rtc_count_config config_rtc_count;
   ```

2. Fill the configuration structure with the default driver configuration.

   ```
   rtc_count_get_config_defaults(&config_rtc_count);
   ```

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

3. Alter the RTC driver configuration to run in 16-bit counting mode, with continuous counter register updates.

   ```
   config_rtc_count.prescaler            = RTC_COUNT_PRESCALER_DIV_1;
   config_rtc_count.mode                 = RTC_COUNT_MODE_16BIT;
   #ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
       config_rtc_count.continuously_update = true;
   #endif
       config_rtc_count.compare_values[0]   = 1000;
   ```

4. Initialize the RTC module.

   ```
   rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
   ```

5. Enable the RTC module, so that it may begin counting.

   ```
   rtc_count_enable(&rtc_instance);
   ```

11.1.3. Implementation

Code used to implement the initialized module.

11.1.3.1. Code

Add after initialization in main().

```
rtc_count_set_period(&rtc_instance, 2000);

while (true) {
    if (rtc_count_is_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0)) {
        /* Do something on RTC count match here */
        port_pin_toggle_output_level(LED_0_PIN);

        rtc_count_clear_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0);
    }
}
```

11.1.3.2. Workflow

1. Set RTC period to 2000ms (two seconds) so that it will overflow and reset back to zero every two seconds.

   ```
   rtc_count_set_period(&rtc_instance, 2000);
   ```

2. Enter an infinite loop to poll the RTC driver to check when a comparison match occurs.

   ```
   while (true) {
   ```

3. Check if the RTC driver has found a match on compare channel 0 against the current RTC count value.

   ```
   if (rtc_count_is_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0)) {
   ```

4. Once a compare match occurs, perform the desired user action.

   ```
   /* Do something on RTC count match here */
   port_pin_toggle_output_level(LED_0_PIN);
   ```

5. Clear the compare match, so that future matches may occur.

   ```
   rtc_count_clear_compare_match(&rtc_instance, RTC_COUNT_COMPARE_0);
   ```

11.2. 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.

11.2.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

11.2.1.1. Clocks and Oscillators

The `conf_clock.h` file needs to be changed with the following values to configure the clocks and oscillators for the module.

The following oscillator settings are needed:

```
/* SYSTEM_CLOCK_SOURCE_OSC32K configuration - Internal 32KHz oscillator */
#  define CONF_CLOCK_OSC32K_ENABLE                true
#  define CONF_CLOCK_OSC32K_STARTUP_TIME          SYSTEM_OSC32K_STARTUP_130
#  define CONF_CLOCK_OSC32K_ENABLE_1KHZ_OUTPUT    true
```

```
#   define CONF_CLOCK_OSC32K_ENABLE_32KHZ_OUTPUT   true
#   define CONF_CLOCK_OSC32K_ON_DEMAND             true
#   define CONF_CLOCK_OSC32K_RUN_IN_STANDBY        false
```

The following generic clock settings are needed:

```
/* Configure GCLK generator 2 (RTC) */
#   define CONF_CLOCK_GCLK_2_ENABLE                true
#   define CONF_CLOCK_GCLK_2_RUN_IN_STANDBY        false
#   define CONF_CLOCK_GCLK_2_CLOCK_SOURCE
SYSTEM_CLOCK_SOURCE_OSC32K
#   define CONF_CLOCK_GCLK_2_PRESCALER             32
#   define CONF_CLOCK_GCLK_2_OUTPUT_ENABLE         false
```

11.2.2. Setup

11.2.2.1. Code

Create an rtc_module struct and add to the main application source file, outside of any functions:

```
struct rtc_module rtc_instance;
```

The following must be added to the user application:

Function for setting up the module:

```
void configure_rtc_count(void)
{
    struct rtc_count_config config_rtc_count;
    rtc_count_get_config_defaults(&config_rtc_count);

    config_rtc_count.prescaler           = RTC_COUNT_PRESCALER_DIV_1;
    config_rtc_count.mode                = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif
    rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

    rtc_count_enable(&rtc_instance);
}
```

Callback function:

```
void rtc_overflow_callback(void)
{
    /* Do something on RTC overflow here */
    port_pin_toggle_output_level(LED_0_PIN);
}
```

Function for setting up the callback functionality of the driver:

```
void configure_rtc_callbacks(void)
{
    rtc_count_register_callback(
            &rtc_instance, rtc_overflow_callback,
RTC_COUNT_CALLBACK_OVERFLOW);
    rtc_count_enable_callback(&rtc_instance, RTC_COUNT_CALLBACK_OVERFLOW);
}
```

Add to user application main():

```
/* Initialize system. Must configure conf_clocks.h first. */
system_init();
```

```
/* Configure and enable RTC */
configure_rtc_count();

/* Configure and enable callback */
configure_rtc_callbacks();

/* Set period */
rtc_count_set_period(&rtc_instance, 2000);
```

11.2.2.2. Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Configure and enable module.

   ```
   configure_rtc_count();
   ```

3. Create an RTC configuration structure to hold the desired RTC driver settings and fill it with the default driver configuration values.

   ```
   struct rtc_count_config config_rtc_count;
   rtc_count_get_config_defaults(&config_rtc_count);
   ```

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

4. Alter the RTC driver configuration to run in 16-bit counting mode, with continuous counter register updates and a compare value of 1000ms.

   ```
   config_rtc_count.prescaler           = RTC_COUNT_PRESCALER_DIV_1;
   config_rtc_count.mode                = RTC_COUNT_MODE_16BIT;
   #ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
       config_rtc_count.continuously_update = true;
   #endif
   ```

5. Initialize the RTC module.

   ```
   rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
   ```

6. Enable the RTC module, so that it may begin counting.

   ```
   rtc_count_enable(&rtc_instance);
   ```

7. Configure callback functionality.

   ```
   configure_rtc_callbacks();
   ```

   1. Register overflow callback.

      ```
      rtc_count_register_callback(
              &rtc_instance, rtc_overflow_callback,
      RTC_COUNT_CALLBACK_OVERFLOW);
      ```

   2. Enable overflow callback.

      ```
      rtc_count_enable_callback(&rtc_instance,
      RTC_COUNT_CALLBACK_OVERFLOW);
      ```

8. Set period.

   ```
   rtc_count_set_period(&rtc_instance, 2000);
   ```

11.2.3. Implementation

11.2.3.1. Code

Add to user application main:

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

11.2.3.2. Workflow

1. Infinite while loop while waiting for callbacks.

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

11.2.4. Callback

Each time the RTC counter overflows, the callback function will be called.

11.2.4.1. 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);
```

11.3. Quick Start Guide for RTC Tamper with DMA

In this use case, the RTC is set up in count mode. The quick start configures the RTC in 32-bit mode and . The rest of the configuration is according to the default. A callback is implemented for when the RTC capture tamper stamp.

11.3.1. Setup

11.3.1.1. Prerequisites

The Generic Clock Generator for the RTC should be configured and enabled; if you are using the System Clock driver, this may be done via `conf_clocks.h`.

11.3.1.2. Code

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

```
struct rtc_module rtc_instance;
```

```
struct dma_resource example_resource;
```

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

The following must be added to the user application: Function for setting up the module:

```
void configure_rtc(void)
{
    struct rtc_count_config config_rtc_count;
    rtc_count_get_config_defaults(&config_rtc_count);
    config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
    rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

    struct rtc_tamper_config config_rtc_tamper;
    rtc_tamper_get_config_defaults(&config_rtc_tamper);
    config_rtc_tamper.dma_tamper_enable = true;
    config_rtc_tamper.in_cfg[0].level = RTC_TAMPER_LEVEL_RISING;
    config_rtc_tamper.in_cfg[0].action = RTC_TAMPER_INPUT_ACTION_CAPTURE;
    rtc_tamper_set_config(&rtc_instance, &config_rtc_tamper);

    rtc_count_enable(&rtc_instance);
}
```

Callback function:

```
void rtc_tamper_callback(void)
{
    /* Do something on RTC tamper capture here */
    LED_On(LED_0_PIN);
}
```

Function for setting up the callback functionality of the driver:

```
void configure_rtc_callbacks(void)
{
    rtc_count_register_callback(
            &rtc_instance, rtc_tamper_callback, RTC_COUNT_CALLBACK_TAMPER);
    rtc_count_enable_callback(&rtc_instance, RTC_COUNT_CALLBACK_TAMPER);
}
```

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

```
/* Initialize system. Must configure conf_clocks.h first. */
system_init();

/* Configure and enable RTC */
configure_rtc();

/* Configure and enable callback */
configure_rtc_callbacks();

configure_dma_resource(&example_resource);

setup_transfer_descriptor(&example_descriptor);

dma_add_descriptor(&example_resource, &example_descriptor);

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

11.3.1.3. Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Configure and enable module.

   ```
   configure_rtc();
   ```

    1. Create a RTC configuration structure to hold the desired RTC driver settings and fill it with the configuration values.

       ```
       struct rtc_count_config config_rtc_count;
       rtc_count_get_config_defaults(&config_rtc_count);
       config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
       rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
       ```

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

       ```
       struct rtc_count_config config_rtc_count;
       rtc_count_get_config_defaults(&config_rtc_count);
       config_rtc_count.prescaler = RTC_COUNT_PRESCALER_DIV_1;
       rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
       ```
    3. Create a RTC tamper configuration structure and fill it with the configuration values.

       ```
       struct rtc_tamper_config config_rtc_tamper;
       rtc_tamper_get_config_defaults(&config_rtc_tamper);
       config_rtc_tamper.dma_tamper_enable = true;
       config_rtc_tamper.in_cfg[0].level = RTC_TAMPER_LEVEL_RISING;
       config_rtc_tamper.in_cfg[0].action =
       RTC_TAMPER_INPUT_ACTION_CAPTURE;
       rtc_tamper_set_config(&rtc_instance, &config_rtc_tamper);
       ```
    4. Enable the RTC module, so that it may begin counting.

       ```
       rtc_count_enable(&rtc_instance);
       ```
3. Configure callback functionality.

   ```
   configure_rtc_callbacks();
   ```

    1. Register overflow callback.

       ```
       rtc_count_register_callback(
               &rtc_instance, rtc_tamper_callback,
       RTC_COUNT_CALLBACK_TAMPER);
       ```
    2. Enable overflow callback.

       ```
       rtc_count_enable_callback(&rtc_instance,
       RTC_COUNT_CALLBACK_TAMPER);
       ```
4. Configure the DMA.
    1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

       ```
       struct dma_resource_config config;
       ```
    2. Initialize the DMA resource configuration struct with the module's. default values.

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

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

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

```
config.peripheral_trigger = RTC_DMAC_ID_TIMESTAMP;
config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
```

4. Allocate a DMA resource with the configurations.

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

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

```
struct dma_descriptor_config descriptor_config;
```

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

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

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

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

```
descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.src_increment_enable = false;
descriptor_config.block_transfer_count = 1;
descriptor_config.source_address = (uint32_t)(&rtc_instance.hw-
>MODE0.TIMESTAMP.reg);
descriptor_config.destination_address = (uint32_t)
(buffer_rtc_tamper);
descriptor_config.next_descriptor_address = (uint32_t)descriptor;
```

8. Create the DMA transfer descriptor.

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

9. Add DMA descriptor to DMA resource.

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

11.3.2. Implementation

11.3.2.1. Code

Add to user application main:

1. Infinite while loop while waiting for callbacks.

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

11.3.2.2. Callback

When the RTC tamper captured, the callback function will be called.

1. LED0 on for RTC tamper capture:

```
/* Do something on RTC tamper capture here */
LED_On(LED_0_PIN);
```

12. Document Revision History

Doc. Rev.	Date	Comments
42111E	12/2015	Added support for SAM L21/L22, SAM C21, SAM D09, and SAM DA1
42111D	12/2014	Added support for SAM R21 and SAM D10/D11
42111C	01/2014	Added support for SAM D21
42111B	06/2013	Added additional documentation on the event system. Corrected documentation typos.
42111A	06/2013	Initial release

Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**