SMART ARM-based Microcontrollers

AT03263: SAM D/R/L/C Timer Counter (TC) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the timer modules within the device, for waveform generation and timing operations. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- TC (Timer/Counter)

The following devices can use this module:

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

The outline of this documentation is as follows:

Table of Contents

1. Software License

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

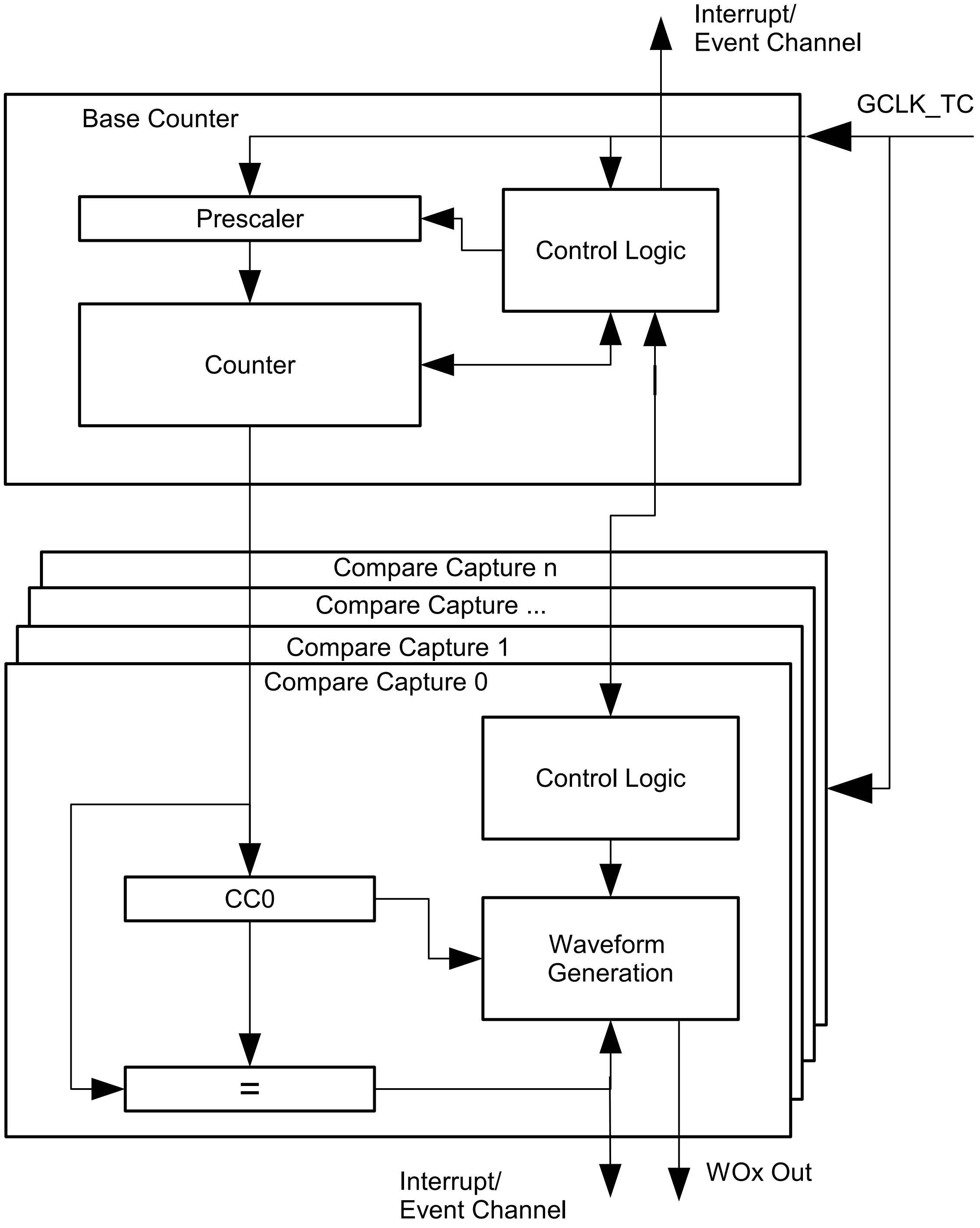
The Timer/Counter (TC) module provides a set of timing and counting related functionality, such as the generation of periodic waveforms, the capturing of a periodic waveform's frequency/duty cycle, and software timekeeping for periodic operations. TC modules can be configured to use an 8-, 16-, or 32-bit counter size.

This TC module for the SAM is capable of the following functions:

- Generation of PWM signals
- Generation of timestamps for events
- General time counting
- Waveform period capture
- Waveform frequency capture

Figure 3-1 Basic Overview of the TC Module on page 8 shows the overview of the TC module design.

Figure 3-1. Basic Overview of the TC Module



3.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_TC_DOUBLE_BUFFERED	SAM L21/L22/C20/C21
FEATURE_TC_SYNCBUSY_SCHEME_VERSION_2	SAM L21/L22/C20/C21
FEATURE_TC_STAMP_PW_CAPTURE	SAM L21/L22/C20/C21
FEATURE_TC_READ_SYNC	SAM L21/L22/C20/C21
FEATURE_TC_IO_CAPTURE	SAM L21/L22/C20/C21
FEATURE_TC_GENERATE_DMA_TRIGGER	SAM L21/L22

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

3.2. Functional Description

Independent of the configured counter size, each TC module can be set up in one of two different modes; capture and compare.

In capture mode, the counter value is stored when a configurable event occurs. This mode can be used to generate timestamps used in event capture, or it can be used for the measurement of a periodic input signal's frequency/duty cycle.

In compare mode, the counter value is compared against one or more of the configured channel compare values. When the counter value coincides with a compare value an action can be taken automatically by the module, such as generating an output event or toggling a pin when used for frequency or Pulse Width Modulation (PWM) signal generation.

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

3.3. Timer/Counter Size

Each timer module can be configured in one of three different counter sizes; 8-, 16-, and 32-bit. The size of the counter determines the maximum value it can count to before an overflow occurs and the count is reset back to zero. Table 3-1 Timer Counter Sizes and Their Maximum Count Values on page 10 shows the maximum values for each of the possible counter sizes.

Table 3-1. Timer Counter Sizes and Their Maximum Count Values

Counter size	Max. (hexadecimal)	Max. (decimal)
8-bit	0xFF	255
16-bit	0xFFFF	65,535
32-bit	0xFFFFFFFF	4,294,967,295

When using the counter in 16- or 32-bit count mode, Compare Capture register 0 (CC0) is used to store the period value when running in PWM generation match mode.

When using 32-bit counter size, two 16-bit counters are chained together in a cascade formation. Except in SAM D09/D10/D11. Even numbered TC modules (e.g. TC0, TC2) can be configured as 32-bit counters. The odd numbered counters will act as slaves to the even numbered masters, and will not be reconfigurable until the master timer is disabled. The pairing of timer modules for 32-bit mode is shown in Table 3-2 TC Master and Slave Module Pairings on page 10.

Table 3-2. TC Master and Slave Module Pairings

Master TC module	Slave TC module
TC0	TC1
TC2	TC3
...	...
TCn-1	TCn

In SAM D09/D10/D11, odd numbered TC modules (e.g. TC1) can be configured as 32-bit counters. The even numbered (e.g. TC2) counters will act as slaves to the odd numbered masters.

3.4. Clock Settings

3.4.1. Clock Selection

Each TC peripheral is clocked asynchronously to the system clock by a GCLK (Generic Clock) channel. The GCLK channel connects to any of the GCLK generators. The GCLK generators are configured to use one of the available clock sources on the system such as internal oscillator, external crystals, etc. See the Generic Clock driver for more information.

3.4.2. Prescaler

Each TC module in the SAM has its own individual clock prescaler, which can be used to divide the input clock frequency used in the counter. This prescaler only scales the clock used to provide clock pulses for the counter to count, and does not affect the digital register interface portion of the module, thus the timer registers will synchronize to the raw GCLK frequency input to the module.

As a result of this, when selecting a GCLK frequency and timer prescaler value the user application should consider both the timer resolution required and the synchronization frequency, to avoid lengthy synchronization times of the module if a very slow GCLK frequency is fed into the TC module. It is preferable to use a higher module GCLK frequency as the input to the timer, and prescale this down as much as possible to obtain a suitable counter frequency in latency-sensitive applications.

3.4.3. Reloading

Timer modules also contain a configurable reload action, used when a re-trigger event occurs. Examples of a re-trigger event are the counter reaching the maximum value when counting up, or when an event from the event system tells the counter to re-trigger. The reload action determines if the prescaler should be reset, and when this should happen. The counter will always be reloaded with the value it is set to start counting from. The user can choose between three different reload actions, described in Table 3-3 TC Module Reload Actions on page 11.

Table 3-3. TC Module Reload Actions

Reload action	Description
TC_RELOAD_ACTION_GCLK	Reload TC counter value on next GCLK cycle. Leave prescaler as-is.
TC_RELOAD_ACTION_PRESC	Reloads TC counter value on next prescaler clock. Leave prescaler as-is.
TC_RELOAD_ACTION_RESYNC	Reload TC counter value on next GCLK cycle. Clear prescaler to zero.

The reload action to use will depend on the specific application being implemented. One example is when an external trigger for a reload occurs; if the TC uses the prescaler, the counter in the prescaler should not have a value between zero and the division factor. The TC counter and the counter in the prescaler should both start at zero. When the counter is set to re-trigger when it reaches the maximum value on the other hand, this is not the right option to use. In such a case it would be better if the prescaler is left unaltered when the re-trigger happens, letting the counter reset on the next GCLK cycle.

3.5. Compare Match Operations

In compare match operation, Compare/Capture registers are used in comparison with the counter value. When the timer's count value matches the value of a compare channel, a user defined action can be taken.

3.5.1. Basic Timer

A Basic Timer is a simple application where compare match operations are used to determine when a specific period has elapsed. In Basic Timer operations, one or more values in the module's Compare/Capture registers are used to specify the time (as a number of prescaled GCLK cycles) when an action should be taken by the microcontroller. This can be an Interrupt Service Routine (ISR), event generator via the event system, or a software flag that is polled via the user application.

3.5.2. Waveform Generation

Waveform generation enables the TC module to generate square waves, or if combined with an external passive low-pass filter; analog waveforms.

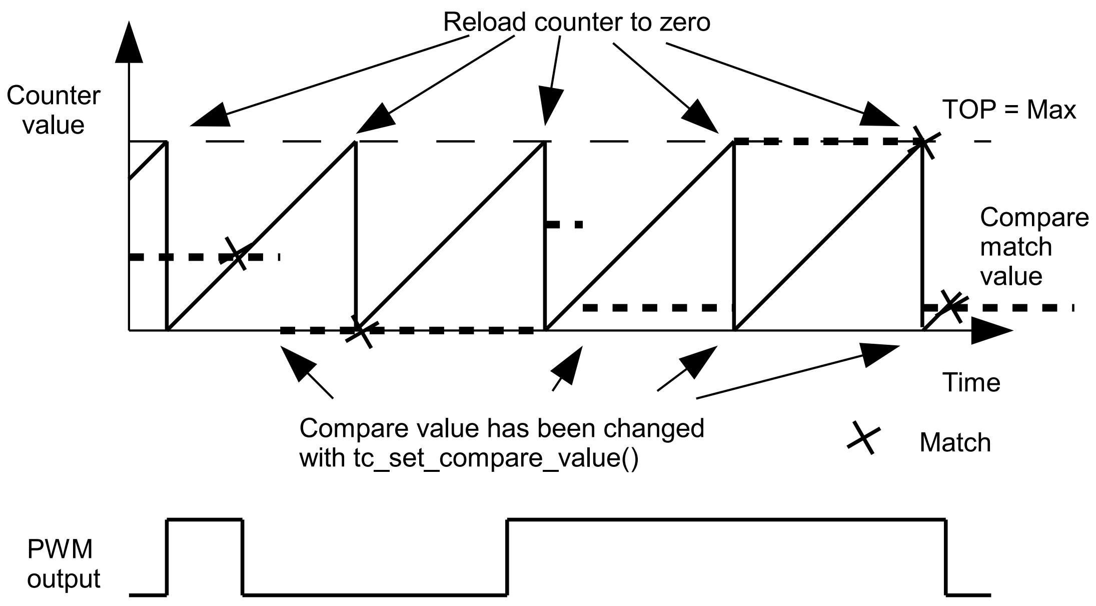
3.5.3. Waveform Generation - PWM

Pulse width modulation is a form of waveform generation and a signalling technique that can be useful in many situations. When PWM mode is used, a digital pulse train with a configurable frequency and duty cycle can be generated by the TC module and output to a GPIO pin of the device.

Often PWM is used to communicate a control or information parameter to an external circuit or component. Differing impedances of the source generator and sink receiver circuits are less of an issue when using PWM compared to using an analog voltage value, as noise will not generally affect the signal's integrity to a meaningful extent.

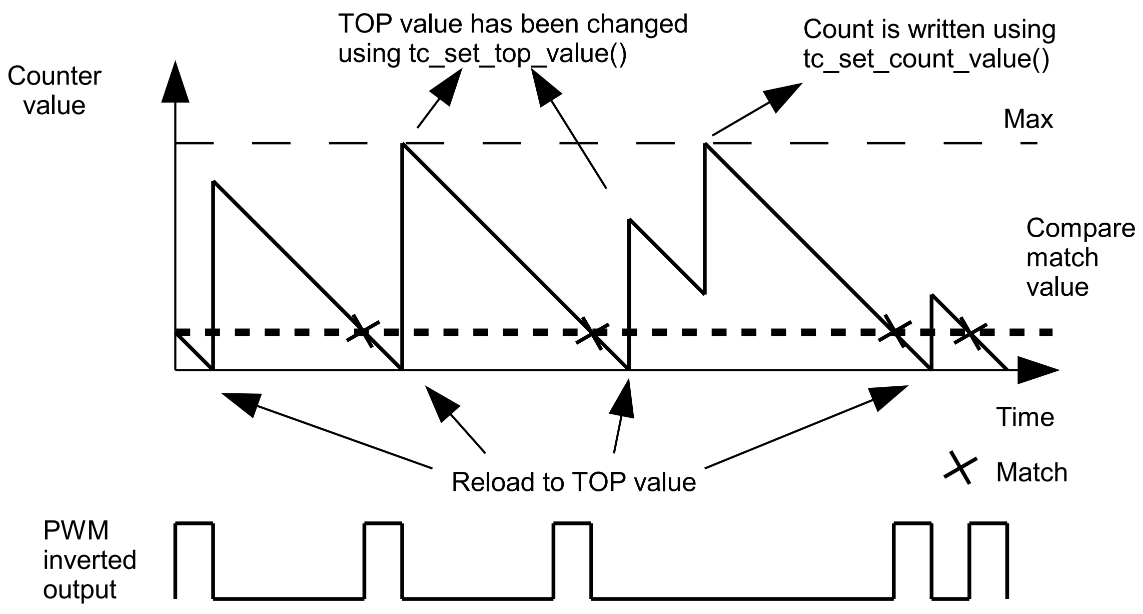
Figure 3-2 Example of PWM in Normal Mode, and Different Counter Operations on page 12 illustrates operations and different states of the counter and its output when running the counter in PWM normal mode. As can be seen, the TOP value is unchanged and is set to MAX. The compare match value is changed at several points to illustrate the resulting waveform output changes. The PWM output is set to normal (i.e. non-inverted) output mode.

Figure 3-2. Example of PWM in Normal Mode, and Different Counter Operations



In Figure 3-3 Example of PWM in Match Mode and Different Counter Operations on page 13, 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 3-3. Example of PWM in Match Mode and Different Counter Operations


Counter value
TOP value has been changed using tc_set_top_value()
Count is written using tc_set_count_value()
Max
Compare match value
Time
Match
Reload to TOP value
PWM inverted output

3.5.4. Waveform Generation - Frequency

Frequency Generation mode is in many ways identical to PWM generation. However, in Frequency Generation a toggle only occurs on the output when a match on a capture channels occurs. When the match is made, the timer value is reset, resulting in a variable frequency square wave with a fixed 50% duty cycle.

3.5.5. Capture Operations

In capture operations, any event from the event system or a pin change can trigger a capture of the counter value. This captured counter value can be used as a timestamp for the event, or it can be used in frequency and pulse width capture.

3.5.6. Capture Operations - Event

Event capture is a simple use of the capture functionality, designed to create timestamps for specific events. When the TC module's input capture pin is externally toggled, the current timer count value is copied into a buffered register which can then be read out by the user application.

Note that when performing any capture operation, there is a risk that the counter reaches its top value (MAX) when counting up, or the bottom value (zero) when counting down, before the capture event occurs. This can distort the result, making event timestamps to appear shorter than reality; the user application should check for timer overflow when reading a capture result in order to detect this situation and perform an appropriate adjustment.

Before checking for a new capture, TC_STATUS_COUNT_OVERFLOW should be checked. The response to an overflow error is left to the user application, however it may be necessary to clear both the capture overflow flag and the capture flag upon each capture reading.

3.5.7. Capture Operations - Pulse Width

Pulse Width Capture mode makes it possible to measure the pulse width and period of PWM signals. This mode uses two capture channels of the counter. This means that the counter module used for Pulse Width Capture can not be used for any other purpose. There are two modes for pulse width capture; Pulse Width Period (PWP) and Period Pulse Width (PPW). In PWP mode, capture channel 0 is used for storing the pulse width and capture channel 1 stores the observed period. While in PPW mode, the roles of the two capture channels are reversed.

As in the above example it is necessary to poll on interrupt flags to see if a new capture has happened and check that a capture overflow error has not occurred.

3.6. One-shot Mode

TC modules can be configured into a one-shot mode. When configured in this manner, starting the timer will cause it to count until the next overflow or underflow condition before automatically halting, waiting to be manually triggered by the user application software or an event signal from the event system.

3.6.1. Wave Generation Output Inversion

The output of the wave generation can be inverted by hardware if desired, resulting in the logically inverted value being output to the configured device GPIO pin.

4. Special Considerations

The number of capture compare registers in each TC module is dependent on the specific SAM device being used, and in some cases the counter size.

The maximum amount of capture compare registers available in any SAM device is two when running in 32-bit mode and four in 8- and 16-bit modes.

5. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

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

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Waveform Inversion Mode

7.1.1.1. Type tc_callback_t

```
typedef void(* tc_callback_t )(struct tc_module *const module)
```

Type of the callback functions.

7.2. Structure Definitions

7.2.1. Struct tc_16bit_config

Table 7-1. Members

Type	Name	Description
uint16_t	compare_capture_channel[]	Value to be used for compare match on each channel
uint16_t	value	Initial timer count value

7.2.2. Struct tc_32bit_config

Table 7-2. Members

Type	Name	Description
uint32_t	compare_capture_channel[]	Value to be used for compare match on each channel
uint32_t	value	Initial timer count value

7.2.3. Struct tc_8bit_config

Table 7-3. Members

Type	Name	Description
uint8_t	compare_capture_channel[]	Value to be used for compare match on each channel
uint8_t	period	Where to count to or from depending on the direction on the counter
uint8_t	value	Initial timer count value

7.2.4. Struct tc_config

Configuration struct for a TC instance. This structure should be initialized by the tc_get_config_defaults function before being modified by the user application.

Table 7-4. Members

Type	Name	Description
union tc_config.@1	@1	Access the different counter size settings through this configuration member.
enum tc_clock_prescaler	clock_prescaler	Specifies the prescaler value for GCLK_TC
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
enum tc_count_direction	count_direction	Specifies the direction for the TC to count
enum tc_counter_size	counter_size	Specifies either 8-, 16-, or 32-bit counter size
bool	double_buffering_enabled	Set to `true` to enable double buffering write. When enabled any write through tc_set_top_value(), tc_set_compare_value() and will direct to the buffer register as buffered value, and the buffered value will be committed to effective register on UPDATE condition, if update is not locked.
bool	enable_capture_on_channel[]	Specifies which channel(s) to enable channel capture operation on
bool	enable_capture_on_IO[]	Specifies which channel(s) to enable I/O capture operation on
bool	oneshot	When `true`, one-shot will stop the TC on next hardware or software re-trigger event or overflow/underflow
struct tc_pwm_channel	pwm_channel[]	Specifies the PWM channel for TC
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
enum tc_wave_generation	wave_generation	Specifies which waveform generation mode to use
uint8_t	waveform_invert_output	Specifies which channel(s) to invert the waveform on. For SAM L21/L22/C20/C21, it's also used to invert I/O input pin.

7.2.5. Union tc_config.__unnamed__

Access the different counter size settings through this configuration member.

Table 7-5. Members

Type	Name	Description
struct tc_16bit_config	counter_16_bit	Struct for 16-bit specific timer configuration
struct tc_32bit_config	counter_32_bit	Struct for 32-bit specific timer configuration
struct tc_8bit_config	counter_8_bit	Struct for 8-bit specific timer configuration

7.2.6. Struct tc_events

Event flags for the tc_enable_events() and tc_disable_events().

Table 7-6. Members

Type	Name	Description
enum tc_event_action	event_action	Specifies which event to trigger if an event is triggered
bool	generate_event_on_compare_channel[]	Generate an output event on a compare channel match
bool	generate_event_on_overflow	Generate an output event on counter overflow
bool	invert_event_input	Specifies if the input event source is inverted, when used in PWP or PPW event action modes
bool	on_event_perform_action	Perform the configured event action when an incoming event is signalled

7.2.7. Struct tc_module

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

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

7.2.8. Struct tc_pwm_channel

Table 7-7. Members

Type	Name	Description
bool	enabled	When `true`, PWM output for the given channel is enabled
uint32_t	pin_mux	Specifies Multiplexer (MUX) setting for each output channel pin
uint32_t	pin_out	Specifies pin output for each channel

7.3. Macro Definitions

7.3.1. Macro FEATURE_TC_DOUBLE_BUFFERED

```
#define FEATURE_TC_DOUBLE_BUFFERED
```

Define port features set according to different device familyTC double buffered.

7.3.2. Macro FEATURE_TC_SYNCBUSY_SCHEME_VERSION_2

```
#define FEATURE_TC_SYNCBUSY_SCHEME_VERSION_2
```

SYNCBUSY scheme version 2.

7.3.3. Macro FEATURE_TC_STAMP_PW_CAPTURE

```
#define FEATURE_TC_STAMP_PW_CAPTURE
```

TC time stamp capture and pulse width capture.

7.3.4. Macro FEATURE_TC_READ_SYNC

```
#define FEATURE_TC_READ_SYNC
```

Read synchronization of COUNT.

7.3.5. Macro FEATURE_TC_IO_CAPTURE

```
#define FEATURE_TC_IO_CAPTURE
```

I/O pin edge capture.

7.3.6. Macro FEATURE_TC_GENERATE_DMA_TRIGGER

```
#define FEATURE_TC_GENERATE_DMA_TRIGGER
```

Generate Direct Memory Access (DMA) triggers.

7.3.7. Module Status Flags

TC status flags, returned by tc_get_status() and cleared by tc_clear_status().

7.3.7.1. Macro TC_STATUS_CHANNEL_0_MATCH

```
#define TC_STATUS_CHANNEL_0_MATCH
```

Timer channel 0 has matched against its compare value, or has captured a new value.

7.3.7.2. Macro TC_STATUS_CHANNEL_1_MATCH

```
#define TC_STATUS_CHANNEL_1_MATCH
```

Timer channel 1 has matched against its compare value, or has captured a new value.

7.3.7.3. Macro TC_STATUS_SYNC_READY

```
#define TC_STATUS_SYNC_READY
```

Timer register synchronization has completed, and the synchronized count value may be read.

7.3.7.4. Macro TC_STATUS_CAPTURE_OVERFLOW

```
#define TC_STATUS_CAPTURE_OVERFLOW
```

A new value was captured before the previous value was read, resulting in lost data.

7.3.7.5. Macro TC_STATUS_COUNT_OVERFLOW

```
#define TC_STATUS_COUNT_OVERFLOW
```

The timer count value has overflowed from its maximum value to its minimum when counting upward, or from its minimum value to its maximum when counting downward.

7.3.7.6. Macro TC_STATUS_CHN0_BUFFER_VALID

```
#define TC_STATUS_CHN0_BUFFER_VALID
```

Channel 0 compare or capture buffer valid.

7.3.7.7. Macro TC_STATUS_CHN1_BUFFER_VALID

```
#define TC_STATUS_CHN1_BUFFER_VALID
```

Channel 1 compare or capture buffer valid.

7.3.7.8. Macro TC_STATUS_PERIOD_BUFFER_VALID

```
#define TC_STATUS_PERIOD_BUFFER_VALID
```

Period buffer valid.

7.3.8. TC Wave Generation Mode

7.3.8.1. Macro TC_WAVE_GENERATION_NORMAL_FREQ_MODE

```
#define TC_WAVE_GENERATION_NORMAL_FREQ_MODE
```

TC wave generation mode: normal frequency.

7.3.8.2. Macro TC_WAVE_GENERATION_MATCH_FREQ_MODE

```
#define TC_WAVE_GENERATION_MATCH_FREQ_MODE
```

TC wave generation mode: match frequency.

7.3.8.3. Macro TC_WAVE_GENERATION_NORMAL_PWM_MODE

```
#define TC_WAVE_GENERATION_NORMAL_PWM_MODE
```

TC wave generation mode: normal PWM.

7.3.8.4. Macro TC_WAVE_GENERATION_MATCH_PWM_MODE

```
#define TC_WAVE_GENERATION_MATCH_PWM_MODE
```

TC wave generation mode: match PWM.

7.3.9. Waveform Inversion Mode

7.3.9.1. Macro TC_WAVEFORM_INVERT_CC0_MODE

```
#define TC_WAVEFORM_INVERT_CC0_MODE
```

Waveform inversion CC0 mode.

7.3.9.2. Macro TC_WAVEFORM_INVERT_CC1_MODE

```
#define TC_WAVEFORM_INVERT_CC1_MODE
```

Waveform inversion CC1 mode.

7.4. Function Definitions

7.4.1. Driver Initialization and Configuration

7.4.1.1. 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 7-8. 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 7-9. Return Values

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

7.4.1.2. Function tc_get_config_defaults()

Initializes config with predefined default values.

```
void tc_get_config_defaults(
        struct tc_config *const config)
```

This function will initialize a given TC configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit counter size on the counter
- No prescaler
- Normal frequency wave generation
- GCLK reload action
- Don't run in standby
- Don't run on demand for SAM L21/L22/C20/C21
- No inversion of waveform output
- No capture enabled
- No I/O capture enabled for SAM L21/L22/C20/C21
- No event input enabled
- Count upward
- Don't perform one-shot operations
- No event action
- No channel 0 PWM output
- No channel 1 PWM output
- Counter starts on 0
- Capture compare channel 0 set to 0
- Capture compare channel 1 set to 0
- No PWM pin output enabled
- Pin and MUX configuration not set
- Double buffer disabled (if have this feature)

Table 7-10. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to a TC module configuration structure to set

7.4.1.3. 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 7-11. Parameters

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

Returns
Status of the initialization procedure.

Table 7-12. 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

7.4.2. Event Management

7.4.2.1. 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 7-13. 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

7.4.2.2. 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 7-14. 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

7.4.3. Enable/Disable/Reset

7.4.3.1. 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 7-15. Parameters

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

Returns
Status of the procedure.

Table 7-16. Return Values

Return value	Description
STATUS_OK	The module was reset successfully
STATUS_ERR_UNSUPPORTED_DEV	A 32-bit slave TC module was passed to the function. Only use reset on master TC

7.4.3.2. 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 7-17. Parameters

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

7.4.3.3. 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 7-18. Parameters

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

7.4.4. Get/Set Count Value

7.4.4.1. 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 7-19. Parameters

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

Returns
Count value of the specified TC module.

7.4.4.2. 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 7-20. 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 7-21. Return Values

Return value	Description
STATUS_OK	The timer count was updated successfully
STATUS_ERR_INVALID_ARG	An invalid timer counter size was specified

7.4.5. Start/Stop Counter

7.4.5.1. Function tc_stop_counter()

Stops the counter.

```
void tc_stop_counter(
        const struct tc_module *const module_inst)
```

This function will stop the counter. When the counter is stopped the value in the count value is set to 0 if the counter was counting up, or maximum if the counter was counting down when stopped.

Table 7-22. Parameters

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

7.4.5.2. 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 7-23. Parameters

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

7.4.6. Double Buffering

7.4.6.1. Function tc_update_double_buffer()

Update double buffer.

```
void tc_update_double_buffer(
        const struct tc_module *const module_inst)
```

Update double buffer.

Table 7-24. Parameters

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

7.4.7. Count Read Synchronization

7.4.7.1. Function tc_sync_read_count()

Read synchronization of COUNT.

```
void tc_sync_read_count(
        const struct tc_module *const module_inst)
```

Read synchronization of COUNT.

Table 7-25. Parameters

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

7.4.8. Generate TC DMA Triggers Command

7.4.8.1. Function tc_dma_trigger_command()

TC DMA Trigger.

```
void tc_dma_trigger_command(
        const struct tc_module *const module_inst)
```

TC DMA trigger command.

Table 7-26. Parameters

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

7.4.9. Get Capture Set Compare

7.4.9.1. 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 7-27. 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.

7.4.9.2. 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 7-28. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	channel_index	Index of the compare channel to write to
[in]	compare	New compare value to set

Returns
Status of the compare update procedure.

Table 7-29. Return Values

Return value	Description
STATUS_OK	The compare value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied

7.4.10. Set Top Value

7.4.10.1. 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 7-30. 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 7-31. 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

7.4.11. Status Management

7.4.11.1. 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 7-32. Parameters

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

Returns
Bitmask of `TC_STATUS_*` flags.

Table 7-33. Return Values

Return value	Description
TC_STATUS_CHANNEL_0_MATCH	Timer channel 0 compare/capture match
TC_STATUS_CHANNEL_1_MATCH	Timer channel 1 compare/capture match
TC_STATUS_SYNC_READY	Timer read synchronization has completed
TC_STATUS_CAPTURE_OVERFLOW	Timer capture data has overflowed
TC_STATUS_COUNT_OVERFLOW	Timer count value has overflowed
TC_STATUS_CHN0_BUFFER_VALID	Timer count channel 0 compare/capture buffer valid
TC_STATUS_CHN1_BUFFER_VALID	Timer count channel 1 compare/capture buffer valid
TC_STATUS_PERIOD_BUFFER_VALID	Timer count period buffer valid

7.4.11.2. 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 7-34. 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

7.5. Enumeration Definitions

7.5.1. Waveform Inversion Mode

7.5.1.1. Enum tc_waveform_invert_output

Output waveform inversion mode.

Table 7-35. 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

7.5.1.2. Enum tc_event_action

Event action to perform when the module is triggered by an event.

Table 7-36. Members

Enum value	Description
TC_EVENT_ACTION_OFF	No event action
TC_EVENT_ACTION_RETRIGGER	Re-trigger on event
TC_EVENT_ACTION_INCREMENT_COUNTER	Increment counter on event
TC_EVENT_ACTION_START	Start counter on event
TC_EVENT_ACTION_PPW	Store period in capture register 0, pulse width in capture register 1
TC_EVENT_ACTION_PWP	Store pulse width in capture register 0, period in capture register 1
TC_EVENT_ACTION_STAMP	Time stamp capture
TC_EVENT_ACTION_PW	Pulse width capture

7.5.2. Enum tc_callback

Enum for the possible callback types for the TC module.

Table 7-37. 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

7.5.3. Enum tc_clock_prescaler

This enum is used to choose the clock prescaler configuration. The prescaler divides the clock frequency of the TC module to make the counter count slower.

Table 7-38. 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

7.5.4. Enum tc_compare_capture_channel

This enum is used to specify which capture/compare channel to do operations on.

Table 7-39. 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

7.5.5. Enum tc_count_direction

Timer/Counter count direction.

Table 7-40. 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

7.5.6. Enum tc_counter_size

This enum specifies the maximum value it is possible to count to.

Table 7-41. Members

Enum value	Description
TC_COUNTER_SIZE_8BIT	The counter's maximum value is 0xFF, the period register is available to be used as top value
TC_COUNTER_SIZE_16BIT	The counter's maximum value is 0xFFFF. There is no separate period register, to modify top one of the capture compare registers has to be used. This limits the amount of available channels.
TC_COUNTER_SIZE_32BIT	The counter's maximum value is 0xFFFFFFFF. There is no separate period register, to modify top one of the capture compare registers has to be used. This limits the amount of available channels.

7.5.7. Enum tc_reload_action

This enum specify how the counter and prescaler should reload.

Table 7-42. 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

7.5.8. Enum tc_wave_generation

This enum is used to select which mode to run the wave generation in.

Table 7-43. Members

Enum value	Description
TC_WAVE_GENERATION_NORMAL_FREQ	Top is maximum, except in 8-bit counter size where it is the PER register
TC_WAVE_GENERATION_MATCH_FREQ	Top is CC0, except in 8-bit counter size where it is the PER register

Enum value	Description
TC_WAVE_GENERATION_NORMAL_PWM	Top is maximum, except in 8-bit counter size where it is the PER register
TC_WAVE_GENERATION_MATCH_PWM	Top is CC0, except in 8-bit counter size where it is the PER register

8. Extra Information for TC Driver

8.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
DMA	Direct Memory Access
TC	Timer Counter
PWM	Pulse Width Modulation
PWP	Pulse Width Period
PPW	Period Pulse Width

8.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

8.3. Errata

There are no errata related to this driver.

8.4. Module History

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

Changelog
Added support for SAM D21 and do some modifications as below: • Clean up in the configuration structure, the counter size setting specific registers is accessed through the counter_8_bit, counter_16_bit, and counter_32_bit structures • All event related settings moved into the tc_event structure
Added automatic digital clock interface enable for the slave TC module when a timer is initialized in 32-bit mode
Initial release

9. Examples for TC Driver

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

- Quick Start Guide for TC - Basic
- Quick Start Guide for TC - Match Frequency Wave Generation
- Quick Start Guide for TC - Timer
- Quick Start Guide for TC - Callback
- Quick Start Guide for Using DMA with TC

9.1. 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

9.1.1. Quick Start

9.1.1.1. Prerequisites

There are no prerequisites for this use case.

9.1.1.2. Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D20 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM R21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D11 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM L21 Xplained Pro.

```
#define PWM_MODULE      EXT2_PWM_MODULE

#define PWM_OUT_PIN     EXT2_PWM_0_PIN

#define PWM_OUT_MUX     EXT2_PWM_0_MUX
```

- SAM L22 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM DA1 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM C21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

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

```
struct tc_module tc_instance;
```

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

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
    config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF / 4);

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);

    tc_enable(&tc_instance);
}
```

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

```
configure_tc();
```

9.1.1.3. Workflow

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

   ```
   struct tc_module tc_instance;
   ```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.
2. Configure the TC module.
   1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

      ```
      struct tc_config config_tc;
      ```
   2. Initialize the TC configuration struct with the module's default values.

      ```
      tc_get_config_defaults(&config_tc);
      ```

      Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
   3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

      ```
      config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
      config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
      config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF /
      4);
      ```
   4. Alter the TC settings to configure the PWM output on a physical device pin.

      ```
      config_tc.pwm_channel[0].enabled = true;
      config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
      config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
      ```
   5. Configure the TC module with the desired settings.

      ```
      tc_init(&tc_instance, PWM_MODULE, &config_tc);
      ```
   6. Enable the TC module to start the timer and begin PWM signal generation.

      ```
      tc_enable(&tc_instance);
      ```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

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

9.1.2.2. Workflow

1. Enter an infinite loop while the PWM wave is generated via the TC module.

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

9.2. Quick Start Guide for TC - Match Frequency Wave Generation

In this use case, the TC will be used to generate a match frequency. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit resolution on the counter
- No prescaler
- Match frequency wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Capture compare channel 0 set to 4000

When system clock is 8MHz, and the compare channel 0 is 4000, the output frequency will be about 1KHz (8000000/4000/2).

9.2.1. Quick Start

9.2.1.1. Prerequisites

There are no prerequisites for this use case.

9.2.1.2. Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro.

  ```
  #define PWM_MODULE      EXT1_PWM_MODULE
  ```

```
#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D20 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

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

```
struct tc_module tc_instance;
```

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

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation = TC_WAVE_GENERATION_MATCH_FREQ;
    config_tc.counter_16_bit.compare_capture_channel[0] = 4000;

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);

    tc_enable(&tc_instance);
}
```

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

```
configure_tc();
```

9.2.1.3. Workflow

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

   ```
   struct tc_module tc_instance;
   ```

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

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

      ```
      struct tc_config config_tc;
      ```

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

      ```
      tc_get_config_defaults(&config_tc);
      ```

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

3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.
   ```
   config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
   config_tc.wave_generation = TC_WAVE_GENERATION_MATCH_FREQ;
   config_tc.counter_16_bit.compare_capture_channel[0] = 4000;
   ```
4. Alter the TC settings to configure the match frequency output on a physical device pin.
   ```
   config_tc.pwm_channel[0].enabled = true;
   config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
   config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
   ```
5. Configure the TC module with the desired settings.
   ```
   tc_init(&tc_instance, PWM_MODULE, &config_tc);
   ```
6. Enable the TC module to start the timer and begin match frequency wave generation.
   ```
   tc_enable(&tc_instance);
   ```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

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

9.2.2.2. Workflow

1. Enter an infinite loop while the match frequency wave is generated via the TC module.
   ```
   while (true) {
       /* Infinite loop */
   }
   ```

9.3. Quick Start Guide for TC - Timer

In this use case, the TC will be used as a timer to generate overflow and compare match callbacks. In the callbacks the on-board LED is toggled.

The TC module will be set up as follows:

- GCLK generator 1 (GCLK 32K) clock source
- 16-bit resolution on the counter
- Prescaler is divided by 64
- GCLK reload action
- Count upward
- Don't run in standby
- No waveform outputs
- No capture enabled
- Don't perform one-shot operations
- No event input enabled
- No event action

- No event generation enabled
- Counter starts on 0
- Counter top set to 2000 (about 4s) and generate overflow callback
- Channel 0 is set to compare and match value 900 and generate callback
- Channel 1 is set to compare and match value 930 and generate callback

9.3.1. Quick Start

9.3.1.1. Prerequisites

For this use case, XOSC32K should be enabled and available through GCLK generator 1 clock source selection. Within Atmel Software Framework (ASF) it can be done through modifying *conf_clocks.h*. See System Clock Management Driver for more details about clock configuration.

9.3.1.2. Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D20 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```
- SAM D21 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```
- SAM R21 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```
- SAM D11 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC1
  ```
- SAM L21 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```
- SAM L22 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```
- SAM DA1 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```
- SAM C21 Xplained Pro.

  ```
  #define CONF_TC_MODULE TC3
  ```

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

```
struct tc_module tc_instance;
```

Copy-paste the following callback function code to your user application:

```
void tc_callback_to_toggle_led(
        struct tc_module *const module_inst)
{
    port_pin_toggle_output_level(LED0_PIN);
}
```

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

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size = TC_COUNTER_SIZE_8BIT;
    config_tc.clock_source = GCLK_GENERATOR_1;
    config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV1024;
    config_tc.counter_8_bit.period = 100;
    config_tc.counter_8_bit.compare_capture_channel[0] = 50;
    config_tc.counter_8_bit.compare_capture_channel[1] = 54;

    tc_init(&tc_instance, CONF_TC_MODULE, &config_tc);

    tc_enable(&tc_instance);
}

void configure_tc_callbacks(void)
{
    tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
            TC_CALLBACK_OVERFLOW);
    tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
            TC_CALLBACK_CC_CHANNEL0);
    tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
            TC_CALLBACK_CC_CHANNEL1);

    tc_enable_callback(&tc_instance, TC_CALLBACK_OVERFLOW);
    tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
    tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL1);
}
```

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

```
configure_tc();
configure_tc_callbacks();
```

9.3.1.3. Workflow

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

   ```
   struct tc_module tc_instance;
   ```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.
2. Configure the TC module.
   1. Create a TC module configuration struct, which can be filled out to adjust the configuration of a physical TC peripheral.

      ```
      struct tc_config config_tc;
      ```
   2. Initialize the TC configuration struct with the module's default values.

      ```
      tc_get_config_defaults(&config_tc);
      ```

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

3. Alter the TC settings to configure the GCLK source, prescaler, period, and compare channel values.

   ```
   config_tc.counter_size = TC_COUNTER_SIZE_8BIT;
   config_tc.clock_source = GCLK_GENERATOR_1;
   config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV1024;
   config_tc.counter_8_bit.period = 100;
   config_tc.counter_8_bit.compare_capture_channel[0] = 50;
   config_tc.counter_8_bit.compare_capture_channel[1] = 54;
   ```

4. Configure the TC module with the desired settings.

   ```
   tc_init(&tc_instance, CONF_TC_MODULE, &config_tc);
   ```

5. Enable the TC module to start the timer.

   ```
   tc_enable(&tc_instance);
   ```

3. Configure the TC callbacks.
   1. Register the Overflow and Compare Channel Match callback functions with the driver.

      ```
      tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
              TC_CALLBACK_OVERFLOW);
      tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
              TC_CALLBACK_CC_CHANNEL0);
      tc_register_callback(&tc_instance, tc_callback_to_toggle_led,
              TC_CALLBACK_CC_CHANNEL1);
      ```

   2. Enable the Overflow and Compare Channel Match callbacks so that it will be called by the driver when appropriate.

      ```
      tc_enable_callback(&tc_instance, TC_CALLBACK_OVERFLOW);
      tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
      tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL1);
      ```

9.3.2. Use Case

9.3.2.1. Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

9.3.2.2. Workflow

1. Enter an infinite loop while the timer is running.

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

9.4. Quick Start Guide for TC - Callback

In this use case, the TC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source

- 16-bit resolution on the counter
- No prescaler
- Normal PWM wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0

9.4.1. Quick Start

9.4.1.1. Prerequisites

There are no prerequisites for this use case.

9.4.1.2. Code

Add to the main application source file, before any functions:

- SAM D21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D20 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM R21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM D11 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM L21 Xplained Pro.

```
#define PWM_MODULE      EXT2_PWM_MODULE
```

```
#define PWM_OUT_PIN     EXT2_PWM_0_PIN

#define PWM_OUT_MUX     EXT2_PWM_0_MUX
```

- SAM L22 Xplained Pro.

```
#define PWM_MODULE      EXT3_PWM_MODULE

#define PWM_OUT_PIN     EXT3_PWM_0_PIN

#define PWM_OUT_MUX     EXT3_PWM_0_MUX
```

- SAM DA1 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

- SAM C21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

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

```
struct tc_module tc_instance;
```

Copy-paste the following callback function code to your user application:

```
void tc_callback_to_change_duty_cycle(
        struct tc_module *const module_inst)
{
    static uint16_t i = 0;

    i += 128;
    tc_set_compare_value(module_inst, TC_COMPARE_CAPTURE_CHANNEL_0, i
+ 1);
}
```

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

```
void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
    config_tc.counter_16_bit.compare_capture_channel[0] = 0xFFFF;

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);

    tc_enable(&tc_instance);
}
```

```
void configure_tc_callbacks(void)
{
    tc_register_callback(
            &tc_instance,
            tc_callback_to_change_duty_cycle,
            TC_CALLBACK_CC_CHANNEL0);

    tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
}
```

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

```
configure_tc();
configure_tc_callbacks();
```

9.4.1.3. Workflow

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

   ```
   struct tc_module tc_instance;
   ```

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

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

      ```
      struct tc_config config_tc;
      ```

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

      ```
      tc_get_config_defaults(&config_tc);
      ```

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

   3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

      ```
      config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
      config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
      config_tc.counter_16_bit.compare_capture_channel[0] = 0xFFFF;
      ```

   4. Alter the TC settings to configure the PWM output on a physical device pin.

      ```
      config_tc.pwm_channel[0].enabled = true;
      config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
      config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
      ```

   5. Configure the TC module with the desired settings.

      ```
      tc_init(&tc_instance, PWM_MODULE, &config_tc);
      ```

   6. Enable the TC module to start the timer and begin PWM signal generation.

      ```
      tc_enable(&tc_instance);
      ```

3. Configure the TC callbacks.
   1. Register the Compare Channel 0 Match callback functions with the driver.

      ```
      tc_register_callback(
              &tc_instance,
      ```

```
        tc_callback_to_change_duty_cycle,
        TC_CALLBACK_CC_CHANNEL0);
```

2. Enable the Compare Channel 0 Match callback so that it will be called by the driver when appropriate.

   ```
   tc_enable_callback(&tc_instance, TC_CALLBACK_CC_CHANNEL0);
   ```

9.4.2. Use Case

9.4.2.1. Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

9.4.2.2. Workflow

1. Enter an infinite loop while the PWM wave is generated via the TC module.

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

9.5. Quick Start Guide for Using DMA with TC

The supported kit list:

- SAM D21/R21/D11/L21/L22/DA1/C21 Xplained Pro

In this use case, the TC will be used to generate a PWM signal. Here the pulse width is set to one quarter of the period. Once the counter value matches the values in the Compare/Capture Value register, an event will be tiggered for a DMA memory to memory transfer. The TC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- 16-bit resolution on the counter
- No prescaler
- Normal PWM wave generation
- GCLK reload action
- Don't run in standby
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Capture compare channel 0 set to 0xFFFF/4

The DMA module is configured for:

- Move data from memory to memory

- Using peripheral trigger of TC6 Match/Compare 0
- Using DMA priority level 0

9.5.1. Quick Start

9.5.1.1. Prerequisites

There are no prerequisites for this use case.

9.5.1.2. Code

Add to the main application source file, before any functions, according to the kit used:

- SAM D21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC6_DMAC_ID_MC_0
```

- SAM R21 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC3_DMAC_ID_MC_0
```

- SAM D11 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN

#define PWM_OUT_MUX     EXT1_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC1_DMAC_ID_MC_0
```

- SAM L21 Xplained Pro.

```
#define PWM_MODULE      EXT2_PWM_MODULE

#define PWM_OUT_PIN     EXT2_PWM_0_PIN

#define PWM_OUT_MUX     EXT2_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC0_DMAC_ID_MC_0
```

- SAM L22 Xplained Pro.

```
#define PWM_MODULE      EXT1_PWM_MODULE

#define PWM_OUT_PIN     EXT1_PWM_0_PIN
```

```
#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC0_DMAC_ID_MC_0
```

- SAM DA1 Xplained Pro.

```
#define PWM_MODULE       EXT1_PWM_MODULE

#define PWM_OUT_PIN      EXT1_PWM_0_PIN

#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC6_DMAC_ID_MC_0
```

- SAM C21 Xplained Pro.

```
#define PWM_MODULE       EXT1_PWM_MODULE

#define PWM_OUT_PIN      EXT1_PWM_0_PIN

#define PWM_OUT_MUX      EXT1_PWM_0_MUX
```

```
#define M2M_DMAC_TRIGGER_ID TC0_DMAC_ID_MC_0
```

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

```
struct tc_module tc_instance;
```

```
struct dma_resource example_resource;
```

```
#define TRANSFER_SIZE    (16)
```

```
#define TRANSFER_COUNTER (32)
```

```
static uint8_t source_memory[TRANSFER_SIZE*TRANSFER_COUNTER];
```

```
static uint8_t destination_memory[TRANSFER_SIZE*TRANSFER_COUNTER];
```

```
static volatile bool transfer_is_done = false;
```

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

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

```
#define TRANSFER_SIZE    (16)

#define TRANSFER_COUNTER (32)

static uint8_t source_memory[TRANSFER_SIZE*TRANSFER_COUNTER];

static uint8_t destination_memory[TRANSFER_SIZE*TRANSFER_COUNTER];

static volatile bool transfer_is_done = false;

COMPILER_ALIGNED(16)
```

```
DmacDescriptor example_descriptor;

void configure_tc(void)
{
    struct tc_config config_tc;
    tc_get_config_defaults(&config_tc);

    config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
    config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
    config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF / 4);

    config_tc.pwm_channel[0].enabled = true;
    config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
    config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;

    tc_init(&tc_instance, PWM_MODULE, &config_tc);

    tc_enable(&tc_instance);
}

void transfer_done(struct dma_resource* const resource )
{
    UNUSED(resource);

    transfer_is_done = true;
}

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

    dma_get_config_defaults(&config);
    config.peripheral_trigger = M2M_DMAC_TRIGGER_ID;

    dma_allocate(resource, &config);
}

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

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = TRANSFER_SIZE;
    descriptor_config.source_address = (uint32_t)source_memory +
TRANSFER_SIZE;
    descriptor_config.destination_address =
            (uint32_t)destination_memory + TRANSFER_SIZE;

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

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

```
configure_tc();
```

9.5.1.3. Workflow

Create variables

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

   ```
   struct tc_module tc_instance;
   ```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.
2. Create a module software instance structure for DMA resource to store the DMA resource state while it is in use.

   ```
   struct dma_resource example_resource;
   ```

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

Configure TC

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

   ```
   struct tc_config config_tc;
   ```
2. Initialize the TC configuration struct with the module's default values.

   ```
   tc_get_config_defaults(&config_tc);
   ```

   Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
3. Alter the TC settings to configure the counter width, wave generation mode, and the compare channel 0 value.

   ```
   config_tc.counter_size    = TC_COUNTER_SIZE_16BIT;
   config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_PWM;
   config_tc.counter_16_bit.compare_capture_channel[0] = (0xFFFF / 4);
   ```
4. Alter the TC settings to configure the PWM output on a physical device pin.

   ```
   config_tc.pwm_channel[0].enabled = true;
   config_tc.pwm_channel[0].pin_out = PWM_OUT_PIN;
   config_tc.pwm_channel[0].pin_mux = PWM_OUT_MUX;
   ```
5. Configure the TC module with the desired settings.

   ```
   tc_init(&tc_instance, PWM_MODULE, &config_tc);
   ```
6. Enable the TC module to start the timer and begin PWM signal generation.

   ```
   tc_enable(&tc_instance);
   ```

Configure DMA

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

   ```
   struct dma_resource_config config;
   ```

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

```
dma_get_config_defaults(&config);
config.peripheral_trigger = M2M_DMAC_TRIGGER_ID;
```

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

3. Allocate a DMA resource with the configurations.

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

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

```
struct dma_descriptor_config descriptor_config;
```

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

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

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

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

```
descriptor_config.block_transfer_count = TRANSFER_SIZE;
descriptor_config.source_address = (uint32_t)source_memory +
TRANSFER_SIZE;
descriptor_config.destination_address =
        (uint32_t)destination_memory + TRANSFER_SIZE;
```

7. Create the DMA transfer descriptor.

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

8. Add the DMA transfer descriptor to the allocated DMA resource.

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

9. Register a callback to indicate transfer status.

```
dma_register_callback(&example_resource, transfer_done,
        DMA_CALLBACK_TRANSFER_DONE);
```

10. The transfer done flag is set in the registered callback function.

```
void transfer_done(struct dma_resource* const resource )
{
    UNUSED(resource);

    transfer_is_done = true;
}
```

Prepare data

1. Setup memory content for validate transfer.

```
for (i = 0; i < TRANSFER_SIZE*TRANSFER_COUNTER; i++) {
    source_memory[i] = i;
}
```

9.5.2. Use Case

9.5.2.1. Code

Copy-paste the following code to your user application:

```
for(i=0;i<TRANSFER_COUNTER;i++) {
    transfer_is_done = false;

    dma_start_transfer_job(&example_resource);

    while (!transfer_is_done) {
        /* Wait for transfer done */
    }

    example_descriptor.SRCADDR.reg += TRANSFER_SIZE;
    example_descriptor.DSTADDR.reg += TRANSFER_SIZE;
}

while(1);
```

9.5.2.2. Workflow

1. Start the loop for transfer.

   ```
   for(i=0;i<TRANSFER_COUNTER;i++) {
       transfer_is_done = false;

       dma_start_transfer_job(&example_resource);

       while (!transfer_is_done) {
           /* Wait for transfer done */
       }

       example_descriptor.SRCADDR.reg += TRANSFER_SIZE;
       example_descriptor.DSTADDR.reg += TRANSFER_SIZE;
   }
   ```

2. Set the transfer done flag as false.

   ```
   transfer_is_done = false;
   ```

3. Start the transfer job.

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

4. Wait for transfer done.

   ```
   while (!transfer_is_done) {
       /* Wait for transfer done */
   }
   ```

5. Update the source and destination address for next transfer.

   ```
   example_descriptor.SRCADDR.reg += TRANSFER_SIZE;
   example_descriptor.DSTADDR.reg += TRANSFER_SIZE;
   ```

6. Enter endless loop.

   ```
   while(1);
   ```

10. Document Revision History

Doc. Rev.	Date	Comments
42123E	12/2015	Added support for SAM L21/L22, SAM DA1, SAM D09, and SAM C21
42123D	12/2014	Added timer use case. Added support for SAM R21 and SAM D10/D11
42123C	01/2014	Added support for SAM D21
42123B	06/2013	Corrected documentation typos
42123A	06/2013	Initial document 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**