APPLICATION NOTE

AT07058: SAM Timer Counter for Control Applications Driver (TCC)

ASF PROGRAMMERS MANUAL

SAM Timer Counter for Control Applications Driver (TCC)

This driver for Atmel® | SMART SAM devices provides an interface for the configuration and management of the TCC module within the device, for waveform generation and timing operations. It also provides extended options for control applications.

The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripherals are used by this module:

- TCC (Timer/Counter for Control Applications)

The following devices can use this module:

- Atmel | SMART SAM D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D10/D11

The outline of this documentation is as follows:

Table of Contents

Software License

1. Prerequisites

There are no prerequisites for this module.

2. Module Overview

The Timer/Counter for Control Applications (TCC) module provides a set of timing and counting related functionality, such as the generation of periodic waveforms, the capturing of a periodic waveform's frequency/duty cycle, software timekeeping for periodic operations, waveform extension control, fault detection etc.

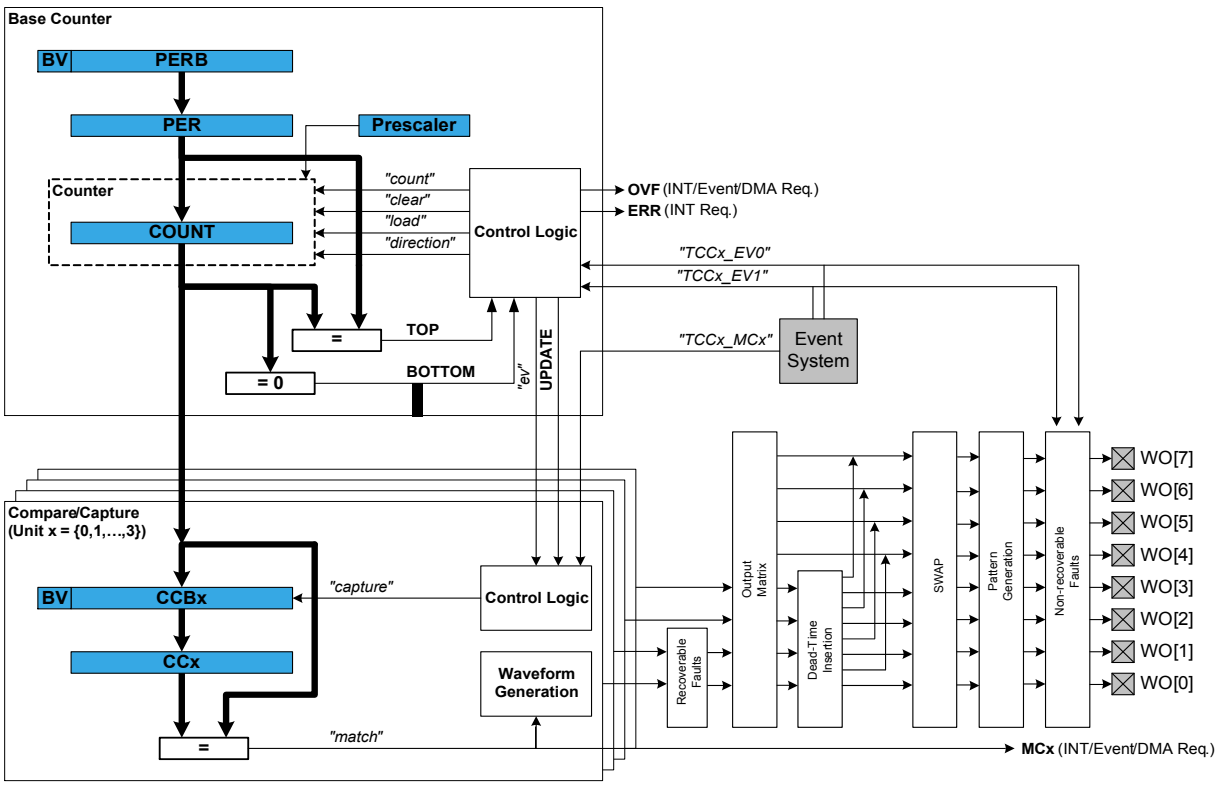
The counter size of the TCC modules can be 16- or 24-bit depending on the TCC instance. Refer SAM TCC Feature List and SAM D10/D11 TCC Feature List for details on TCC instances.

The TCC module for the SAM includes the following functions:

- Generation of PWM signals
- Generation of timestamps for events
- General time counting
- Waveform period capture
- Waveform frequency capture
- Additional control for generated waveform outputs
- Fault protection for waveform generation

Figure 2-1: Overview of the TCC Module on page 7 shows the overview of the TCC Module.

Figure 2-1. Overview of the TCC Module



2.1 Functional Description

The TCC module consists of following sections:

- Base Counter
- Compare/Capture channels, with waveform generation

- Waveform extension control and fault detection
- Interface to the event system, DMAC, and the interrupt system

The base counter can be configured to either count a prescaled generic clock or events from the event system. (TCEx, with event action configured to counting). The counter value can be used by compare/capture channels which can be set up either in compare mode or capture mode.

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

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

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

In compare mode, when output signal is generated, extended waveform controls are available, to arrange the compare outputs into specific formats. The Output matrix can change the channel output routing. Pattern generation unit can overwrite the output signal line to specific state. The Fault protection feature of the TCC supports recoverable and non-recoverable faults.

2.2 Base Timer/Counter

2.2.1 Timer/Counter Size

Each TCC has a counter size of either 16- or 24-bits. The size of the counter determines the maximum value it can count to before an overflow occurs. Table 2-1: Timer Counter Sizes and Their Maximum Count Values on page 8 shows the maximum values for each of the possible counter sizes.

Table 2-1. Timer Counter Sizes and Their Maximum Count Values

Counter size	Max. (hexadecimal)	Max. (decimal)
16-bit	0xFFFF	65,535
24-bit	0xFFFFFF	16,777,215

The period/top value of the counter can be set, to define counting period. This will allow the counter to overflow when the counter value reaches the period/top value.

2.2.2 Timer/Counter Clock and Prescaler

TCC is clocked asynchronously to the system clock by a GCLK (Generic Clock) channel. The GCLK channel can be connected to any of the GCLK generators. The GCLK generators are configured to use one of the available clock sources in the system such as internal oscillator, external crystals, etc. - see the Generic Clock driver for more information.

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

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

2.2.3 Timer/Counter Control Inputs (Events)

The TCC can take several actions on the occurrence of an input event. The event actions are listed in Table 2-2: TCC Module Event Actions on page 9.

Table 2-2. TCC Module Event Actions

Event action	Description	Applied event
TCC_EVENT_ACTION_OFF	No action on the event input	All
TCC_EVENT_ACTION_RETRIGGEF	Re-trigger Counter on event	All
TCC_EVENT_ACTION_NON_RECO	Generate Non-Recoverable Fault on event	All
TCC_EVENT_ACTION_START	Counter start on event	EV0
TCC_EVENT_ACTION_DIR_CONTF	Counter direction control	EV0
TCC_EVENT_ACTION_DECREMEN	Counter decrement on event	EV0
TCC_EVENT_ACTION_PERIOD_PU	Capture pulse period and pulse width	EV0
TCC_EVENT_ACTION_PULSE_WID	Capture pulse width and pulse period	EV0
TCC_EVENT_ACTION_STOP	Counter stop on event	EV1
TCC_EVENT_ACTION_COUNT_EV	Counter count on event	EV1
TCC_EVENT_ACTION_INCREMEN	Counter increment on event	EV1
TCC_EVENT_ACTION_COUNT_DU	Counter count during active state of asynchronous event	EV1

2.2.4 Timer/Counter Reloading

The TCC also has a configurable reload action, used when a re-trigger event occurs. Examples of a re-trigger event could be the counter reaching the maximum value when counting up, or when an event from the event system makes the counter to re-trigger. The reload action determines if the prescaler should be reset, and on which clock. The counter will always be reloaded with the value it is set to start counting. The user can choose between three different reload actions, described in Table 2-3: TCC Module Reload Actions on page 9.

Table 2-3. TCC Module Reload Actions

Reload action	Description
TCC_RELOAD_ACTION_GCLK	Reload TCC counter value on next GCLK cycle. Leave prescaler as-is.
TCC_RELOAD_ACTION_PRESC	Reloads TCC counter value on next prescaler clock. Leave prescaler as-is.
TCC_RELOAD_ACTION_RESYNC	Reload TCC counter value on next GCLK cycle. Clear prescaler to zero.

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

2.2.5 One-shot Mode

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

2.3 Capture Operations

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

2.3.1 Capture Operations - Event

Event capture is a simple use of the capture functionality, designed to create timestamps for specific events. When the input event appears, the current counter value is copied into the corresponding compare/capture register, which can then be read by the user application.

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

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

2.3.2 Capture Operations - Pulse Width

Pulse Width Capture mode makes it possible to measure the pulse width and period of PWM signals. This mode uses two capture channels of the counter. There are two modes for pulse width capture; Pulse Width Period (PWP) and Period Pulse Width (PPW). In PWP mode, capture channel 0 is used for storing the pulse width and capture channel 1 stores the observed period. While in PPW mode, the roles of the two capture channels are reversed.

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

Refer to Timer/Counter Control Inputs (Events) to set up the input event to perform pulse width capture.

2.4 Compare Match Operation

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

2.4.1 Basic Timer

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

2.4.2 Waveform Generation

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

2.4.3 Waveform Generation - PWM

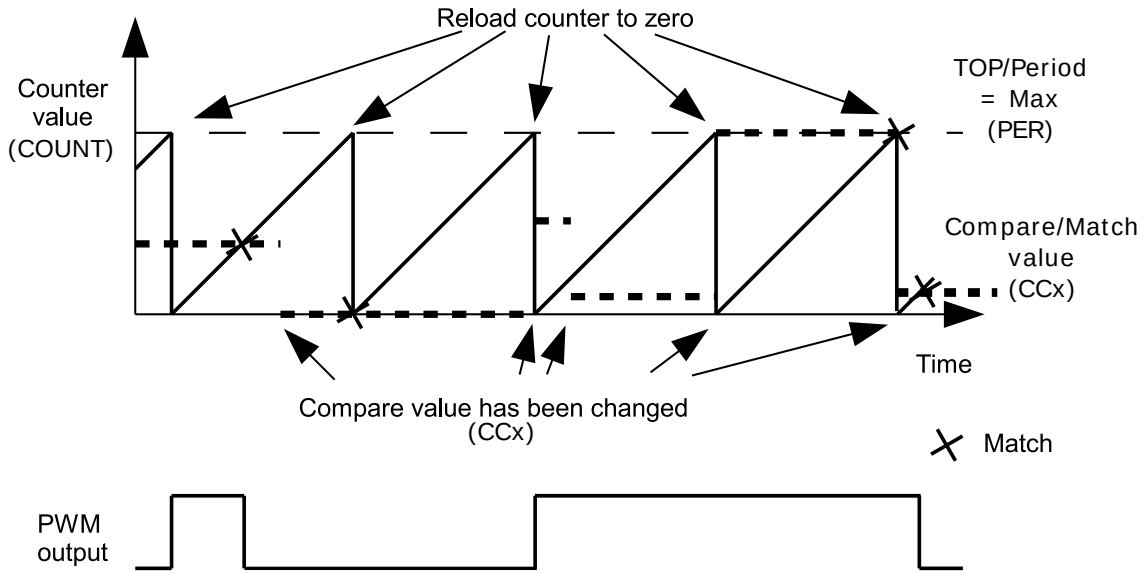
Pulse width modulation is a form of waveform generation and a signalling technique that can be useful in many applications. When PWM mode is used, a digital pulse train with a configurable frequency and duty cycle can be generated by the TCC module and output to a GPIO pin of the device.

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

Figure 2-2: Example Of PWM In Single-Slope Mode, and Different Counter Operations on page 11 illustrates operations and different states of the counter and its output when using the timer in Normal PWM mode (Single

Slope). As can be seen, the TOP/PERIOD value is unchanged and is set to MAX. The compare match value is changed at several points to illustrate the resulting waveform output changes. The PWM output is set to normal (i.e. non-inverted) output mode.

Figure 2-2. Example Of PWM In Single-Slope Mode, and Different Counter Operations



Several PWM modes are supported by the TCC module, refer to datasheet for the details on PWM waveform generation.

2.4.4 Waveform Generation - Frequency

Normal Frequency Generation is in many ways identical to PWM generation. However, only in Frequency Generation, a toggle occurs on the output when a match on a compare channels occurs.

When the Match Frequency Generation is used, the timer value is reset on match condition, resulting in a variable frequency square wave with a fixed 50% duty cycle.

2.5 Waveform Extended Controls

2.5.1 Pattern Generation

Pattern insertion allows the TCC module to change the actual pin output level without modifying the compare/match settings.

Table 2-4. TCC Module Output Pattern Generation

Pattern	Description
TCC_OUTPUT_PATTERN_DISABLE	Pattern disabled, generate output as is
TCC_OUTPUT_PATTERN_0	Generate pattern 0 on output (keep the output LOW)
TCC_OUTPUT_PATTERN_1	Generate pattern 1 on output (keep the output HIGH)

2.5.2 Recoverable Faults

The recoverable faults can trigger one or several of following fault actions:

1. *Halt* action: The recoverable faults can halt the TCC timer/counter, so that the final output wave is kept at a defined state. When the fault state is removed it is possible to recover the counter and waveform generation. The halt action is defined as:

Table 2-5. TCC Module Recoverable Fault Halt Actions

Action	Description
TCC_FAULT_HALT_ACTION_DISABLE	Halt action is disabled
TCC_FAULT_HALT_ACTION_HW_HALT	The timer/counter is halted as long as the corresponding fault is present
TCC_FAULT_HALT_ACTION_SW_HALT	The timer/counter is halted until the corresponding fault is removed and fault state cleared by software
TCC_FAULT_HALT_ACTION_NON_RECOVERABLE	Force all the TCC output pins to a pre-defined level, as what Non-Recoverable Fault do

2. *Restart* action: When enabled, the recoverable faults can restart the TCC timer/counter.
3. *Keep* action: When enabled, the recoverable faults can keep the corresponding channel output to zero when the fault condition is present.
4. *Capture* action: When the recoverable fault occurs, the capture action can time stamps the corresponding fault. The following capture mode is supported:

Table 2-6. TCC Module Recoverable Fault Capture Actions

Action	Description
TCC_FAULT_CAPTURE_DISABLE	Capture action is disabled
TCC_FAULT_CAPTURE_EACH	Equivalent to standard capture operation, on each fault occurrence the time stamp is captured
TCC_FAULT_CAPTURE_MINIMUM	Get the minimum time stamped value in all time stamps
TCC_FAULT_CAPTURE_MAXIMUM	Get the maximum time stamped value in all time stamps
TCC_FAULT_CAPTURE_SMALLER	Time stamp the fault input if the value is smaller than last one
TCC_FAULT_CAPTURE_BIGGER	Time stamp the fault input if the value is bigger than last one
TCC_FAULT_CAPTURE_CHANGE	Time stamp the fault input if the time stamps changes its increment direction

In TCC module, only the first two compare channels (CC0 and CC1) can work with recoverable fault inputs. The corresponding event inputs (TCCx MC0 and TCCx MC1) are then used as fault inputs respectively. The faults are called Fault A and Fault B.

The recoverable fault can be filtered or effected by corresponding channel output. On fault condition there are many other settings that can be chosen. Refer to data sheet for more details about the recoverable fault operations.

2.5.3 Non-Recoverable Faults

The non-recoverable faults force all the TCC output pins to a pre-defined level (can be forced to 0 or 1).
The input control signal of non-recoverable fault is from timer/counter event (TCCx EV0 and TCCx EV1). To enable non-recoverable fault, corresponding TCEx event action must be set to non-recoverable fault action (TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT on page 43). Refer to Timer/Counter Control Inputs (Events) to see the available event input action.

2.6 Double and Circular Buffering

The pattern, period and the compare channels registers are double buffered. For these options there are effective registers (PATT, PER, and CCx) and buffer registers (PATTB, PERB, and CCx). When writing to the buffer registers, the values are buffered and will be committed to effective registers on UPDATE condition.

Usually the buffered value is cleared after it's committed, but there is also option to circular the register buffers. The period (PER) and four lowest compare channels register (CCx, x is 0 ~ 3) support this function. When circular buffer is used, on UPDATE the previous period or compare values are copied back into the corresponding period buffer and compare buffers. This way, the register value and its buffer register value is actually switched on UPDATE condition, and will be switched back on next UPDATE condition.

For input capture, the buffer register (CCBx) and the corresponding capture channel register (CCx) act like a FIFO. When regular register (CCx) is empty or read, any content in the buffer register is passed to regular one.

In TCC module driver, when the double buffering write is enabled, any write through tcc_set_top_value(), tcc_set_compare_value(), and tcc_set_pattern() will be done to the corresponding buffer register. Then the value in the buffer register will be transferred to the regular register on the next UPDATE condition or by a force UPDATE using tcc_force_double_buffer_update().

2.7 Sleep Mode

TCC modules can be configured to operate in any sleep mode, with its "run in standby" function enabled. It can wake up the device using interrupts or perform internal actions with the help of the Event System.

3. Special Considerations

3.1 Module Features

The features of TCC, such as timer/counter size, number of compare capture channels, and number of outputs, are dependent on the TCC module instance being used.

3.1.1 SAM TCC Feature List

For SAM D21/R21, the TCC features are:

Table 3-1. TCC Module Features For SAM D21/R21

TCC#	Match/ Capture channels	Wave outputs	Counter size [bits]	Fault	Dithering	Output matrix	Dead-Time insertion	SWAP	Pattern
0	4	8	24	Y	Y	Y	Y	Y	Y
1	2	4	24	Y	Y				Y
2	2	2	16	Y					

3.1.2 SAM D10/D11 TCC Feature List

For SAM D10/D11, the TCC features are:

Table 3-2. TCC Module Features For SAM D10/D11

TCC#	Match/ Capture channels	Wave outputs	Counter size [bits]	Fault	Dithering	Output matrix	Dead-Time insertion	SWAP	Pattern
0	4	8	24	Y	Y	Y	Y	Y	Y

3.2 Channels vs. Pin outs

As the TCC module may have more waveform output pins than the number of compare/capture channels, the free pins (with number higher than number of channels) will reuse the waveform generated by channels subsequently. E.g., if the number of channels is four and the number of wave output pins is eight, channel 0 output will be available on out pin 0 and 4, channel 1 output on wave out pin 1 and 5, and so on.

4. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

5. Examples

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

6. API Overview

6.1 Variable and Type Definitions

6.1.1 Type tcc_callback_t

```
typedef void(* tcc_callback_t )(struct tcc_module *const module)
```

Type definition for the TCC callback function.

6.2 Structure Definitions

6.2.1 Struct tcc_capture_config

Structure used when configuring TCC channels in capture mode.

Table 6-1. Members

Type	Name	Description
enum tcc_channel_function	channel_function[]	Channel functions selection (capture/match).

6.2.2 Struct tcc_config

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

Table 6-2. Members

Type	Name	Description
union tcc_config.@1	@1	TCC match/capture configurations.
struct tcc_counter_config	counter	Structure for configuring TCC base timer/counter.
bool	double_buffering_enabled	Set to true to enable double buffering write. When enabled any write through tcc_set_top_value(), tcc_set_compare_value() and tcc_set_pattern() will direct to the buffer register as buffered value, and the buffered value will be committed to effective register on UPDATE condition, if update is not locked.[1]
struct tcc_pins_config	pins	Structure for configuring TCC output pins.
bool	run_in_standby	When true the module is enabled during standby.
struct tcc_wave_extension_config	wave_ext	Structure for configuring TCC waveform extension.

Notes: [1]The init values in tcc_config for tcc_init are always filled to effective registers, no matter double buffering enabled or not.

6.2.3 Union tcc_config.__unnamed__

TCC match/capture configurations.

Table 6-3. Members

Type	Name	Description
struct tcc_capture_config	capture	Helps to configure a TCC channel in capture mode.
struct tcc_match_wave_config	compare	For configuring a TCC channel in compare mode.
struct tcc_match_wave_config	wave	Serves the same purpose as compare. Used as an alias for compare, when a TCC channel is configured for wave generation.

6.2.4 Struct tcc_counter_config

Structure for configuring a TCC as a counter.

Table 6-4. Members

Type	Name	Description
enum tcc_clock_prescaler	clock_prescaler	Specifies the prescaler value for GCLK_TCC.
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral.
uint32_t	count	Value to initialize the count register.
enum tcc_count_direction	direction	Specifies the direction for the TCC to count.
bool	oneshot	When true, counter will be stopped on the next hardware or software re-trigger event or overflow/ underflow.
uint32_t	period	Period/top and period/top buffer values for counter.
enum tcc_reload_action	reload_action	Specifies the reload or reset time of the counter and prescaler resynchronization on a re-trigger event for the TCC.

6.2.5 Struct tcc_events

Event flags for the tcc_enable_events() and tcc_disable_events().

Table 6-5. Members

Type	Name	Description
bool	generate_event_on_channel[]	Generate an output event on a channel capture/match. Specify which channels will generate events.

Type	Name	Description
bool	generate_event_on_counter_event	Generate an output event on counter boundary. See tcc_event_output_action.
bool	generate_event_on_counter_overflow	Generate an output event on counter overflow/underflow.
bool	generate_event_on_counter_retrigge	Generate an output event on counter retrigger.
struct tcc_input_event_config	input_config[]	Input events configuration.
bool	on_event_perform_channel_action[]	Perform the configured event action when an incoming channel event is signalled.
bool	on_input_event_perform_action[]	Perform the configured event action when an incoming event is signalled.
struct tcc_output_event_config	output_config	Output event configuration.

6.2.6 Struct tcc_input_event_config

For configuring an input event.

Table 6-6. Members

Type	Name	Description
enum tcc_event_action	action	Event action on incoming event.
bool	invert	Invert incoming event input line.
bool	modify_action	Modify event action.

6.2.7 Struct tcc_match_wave_config

The structure, which helps to configure a TCC channel for compare operation and wave generation.

Table 6-7. Members

Type	Name	Description
enum tcc_channel_function	channel_function[]	Channel functions selection (capture/match).
uint32_t	match[]	Value to be used for compare match on each channel.
enum tcc_wave_generation	wave_generation	Specifies which waveform generation mode to use.
enum tcc_wave_polarity	wave_polarity[]	Specifies polarity for match output waveform generation.
enum tcc_ramp	wave_ramp	Specifies Ramp mode for waveform generation.

6.2.8 Struct tcc_module

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

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

Table 6-8. Members

Type	Name	Description
tcc_callback_t	callback[]	Array of callbacks.
bool	double_buffering_enabled	Set to true to write to buffered registers.
uint32_t	enable_callback_mask	Bit mask for callbacks enabled.
Tcc *	hw	Hardware module pointer of the associated Timer/Counter peripheral.
uint32_t	register_callback_mask	Bit mask for callbacks registered.

6.2.9 Struct tcc_non_recoverable_fault_config

Table 6-9. Members

Type	Name	Description
uint8_t	filter_value	Fault filter value applied on TCEx event input line (0x0 ~ 0xF). Must be 0 when TCEx event is used as synchronous event.
enum tcc_fault_state_output	output	Output.

6.2.10 Struct tcc_output_event_config

Structure used for configuring an output event.

Table 6-10. Members

Type	Name	Description
enum tcc_event_generation_selection	generation_selection	It decides which part of the counter cycle the counter event output is generated.
bool	modify_generation_selection	A switch to allow enable/disable of events, without modifying the event output configuration.

6.2.11 Struct tcc_pins_config

Structure which is used when taking wave output from TCC.

Table 6-11. Members

Type	Name	Description
bool	enable_wave_out_pin[]	When true, PWM output pin for the given channel is enabled.

Type	Name	Description
uint32_t	wave_out_pin[]	Specifies pin output for each channel.
uint32_t	wave_out_pin_mux[]	Specifies MUX setting for each output channel pin.

6.2.12 Struct tcc_recoverable_fault_config

Table 6-12. Members

Type	Name	Description
enum tcc_fault_blanking	blanking	Fault Blanking Start Point for recoverable Fault.
uint8_t	blanking_cycles	Fault blanking value (0 ~ 255), disable input source for several TCC clocks after the detection of the waveform edge.
enum tcc_fault_capture_action	capture_action	Capture action for recoverable Fault.
enum tcc_fault_capture_channel	capture_channel	Channel triggered by recoverable Fault.
uint8_t	filter_value	Fault filter value applied on MCEx event input line (0x0 ~ 0xF). Must be 0 when MCEx event is used as synchronous event. Apply to both recoverable and non-recoverable fault.
enum tcc_fault_halt_action	halt_action	Halt action for recoverable Fault.
bool	keep	Set to true to enable keep action (keep until end of TCC cycle).
bool	qualification	Set to true to enable input qualification (disable input when output is inactive).
bool	restart	Set to true to enable restart action.
enum tcc_fault_source	source	Specifies if the event input generates recoverable Fault. The event system channel connected to MCEx event input must be configured as asynchronous.

6.2.13 Struct tcc_wave_extension_config

This structure is used to specify the waveform extension features for TCC.

Table 6-13. Members

Type	Name	Description
bool	invert[]	Invert waveform final outputs lines.
struct tcc_non_recoverable_fault_config	non_recoverable_fault[]	Configuration for non-recoverable faults.

Type	Name	Description
struct tcc_recoverable_fault_config	recoverable_fault[]	Configuration for recoverable faults.

6.3 Macro Definitions

6.3.1 Module Status Flags

TCC status flags, returned by tcc_get_status() and cleared by tcc_clear_status().

6.3.1.1 Macro TCC_STATUS_CHANNEL_MATCH_CAPTURE

```
#define TCC_STATUS_CHANNEL_MATCH_CAPTURE(ch) \
  (1UL << (ch))
```

Timer channel ch (0 ~ 3) has matched against its compare value, or has captured a new value.

6.3.1.2 Macro TCC_STATUS_CHANNEL_OUTPUT

```
#define TCC_STATUS_CHANNEL_OUTPUT(ch) \
  (1UL << ((ch)+8))
```

Timer channel ch (0 ~ 3) match/compare output state.

6.3.1.3 Macro TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR

```
#define TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(x) \
  (1UL << ((x)+16))
```

A Non-Recoverable Fault x (0 ~ 1) has occurred.

6.3.1.4 Macro TCC_STATUS_RECOVERABLE_FAULT_OCCUR

```
#define TCC_STATUS_RECOVERABLE_FAULT_OCCUR(n) \
  (1UL << ((n)+18))
```

A Recoverable Fault n (0 ~ 1 representing A ~ B) has occured.

6.3.1.5 Macro TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT

```
#define TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT(x) \
  (1UL << ((x)+20))
```

The Non-Recoverable Fault x (0 ~ 1) input is present.

6.3.1.6 Macro TCC_STATUS_RECOVERABLE_FAULT_PRESENT

```
#define TCC_STATUS_RECOVERABLE_FAULT_PRESENT(n) \
  (1UL << ((n)+22))
```

A Recoverable Fault n (0 ~ 1 representing A ~ B) is present.

6.3.1.7 Macro TCC_STATUS_SYNC_READY

```
#define TCC_STATUS_SYNC_READY (1UL << 23)
```

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

6.3.1.8 Macro TCC_STATUS_CAPTURE_OVERFLOW

```
#define TCC_STATUS_CAPTURE_OVERFLOW (1UL << 24)
```

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

6.3.1.9 Macro TCC_STATUS_COUNTER_EVENT

```
#define TCC_STATUS_COUNTER_EVENT (1UL << 25)
```

A counter event occurred.

6.3.1.10 Macro TCC_STATUS_COUNTER_RETRIGGERED

```
#define TCC_STATUS_COUNTER_RETRIGGERED (1UL << 26)
```

A counter retrigger occurred.

6.3.1.11 Macro TCC_STATUS_COUNT_OVERFLOW

```
#define TCC_STATUS_COUNT_OVERFLOW (1UL << 27)
```

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.

6.3.1.12 Macro TCC_STATUS_RAMP_CYCLE_INDEX

```
#define TCC_STATUS_RAMP_CYCLE_INDEX (1UL << 28)
```

Ramp period cycle index. In ramp operation, each two period cycles are marked as cycle A and B, the index 0 represents cycle A and 1 represents cycle B.

6.3.1.13 Macro TCC_STATUS_STOPPED

```
#define TCC_STATUS_STOPPED (1UL << 29)
```

The counter has been stopped (due to disable, stop command or one-shot).

6.3.2 Macro _TCC_CHANNEL_ENUM_LIST

```
#define _TCC_CHANNEL_ENUM_LIST(type) \
  MREPEAT(TCC_NUM_CHANNELS, _TCC_ENUM, type##_CHANNEL)
```

6.3.3 Macro _TCC_ENUM

```
#define _TCC_ENUM(n, type) \
  TCC_##type##_##n,
```

6.3.4 Macro _TCC_WO_ENUM_LIST

```
#define _TCC_WO_ENUM_LIST(type) \
  MREPEAT(TCC_NUM_WAVE_OUTPUTS, _TCC_ENUM, type)
```

6.3.5 Macro TCC_NUM_CHANNELS

```
#define TCC_NUM_CHANNELS 4
```

Maximum number of channels supported by the driver (Channel index from 0 to TCC_NUM_CHANNELS - 1).

6.3.6 Macro TCC_NUM_FAULTS

```
#define TCC_NUM_FAULTS 2
```

Maximum number of (recoverable) faults supported by the driver.

6.3.7 Macro TCC_NUM_WAVE_OUTPUTS

```
#define TCC_NUM_WAVE_OUTPUTS 8
```

Maximum number of wave outputs lines supported by the driver (Output line index from 0 to TCC_NUM_WAVE_OUTPUTS - 1).

6.4 Function Definitions

6.4.1 Driver Initialization and Configuration

6.4.1.1 Function tcc_is_syncing()

Determines if the hardware module is currently synchronizing to the bus.

```
bool tcc_is_syncing(
  const struct tcc_module *const module_inst)
```

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

Table 6-14. Parameters

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

Returns

Synchronization status of the underlying hardware module.

Table 6-15. Return Values

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

6.4.1.2 Function tcc_get_config_defaults()

Initializes config with predefined default values.

```
void tcc_get_config_defaults(
  struct tcc_config *const config,
  Tcc *const hw)
```

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

The default configuration is as follows:

- Don't run in standby
- When setting top,compare or pattern by API, do double buffering write
- The base timer/counter configurations:
  - GCLK generator 0 clock source
  - No prescaler
  - GCLK reload action

- Count upward
- Don't perform one-shot operations
- Counter starts on 0
- Period/top value set to maximum

- The match/capture configurations:
  - All Capture compare channel value set to 0
  - No capture enabled (all channels use compare function)
  - Normal frequency wave generation
  - Waveform generation polarity set to 0
  - Don't perform ramp on waveform
- The waveform extension configurations:
  - No recoverable fault is enabled, fault actions are disabled, filter is set to 0
  - No non-recoverable fault state output is enabled and filter is 0
  - No inversion of waveform output
- No channel output enabled
- No PWM pin output enabled
- Pin and MUX configuration not set

Table 6-16. Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to a TCC module configuration structure to set
[in]	hw	Pointer to the TCC hardware module

6.4.1.3 Function tcc_init()

Initializes a hardware TCC module instance.

```
enum status_code tcc_init(
  struct tcc_module *const module_inst,
  Tcc *const hw,
  const struct tcc_config *const config)
```

Enables the clock and initializes the given TCC module, based on the given configuration values.

Table 6-17. Parameters

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

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

Returns Status of the initialization procedure.

Table 6-18. 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

6.4.2 Event Management

6.4.2.1 Function tcc_enable_events()

Enables the TCC module event input or output.

```
enum status_code tcc_enable_events(
  struct tcc_module *const module_inst,
  struct tcc_events *const events)
```

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

Note Events cannot be altered while the module is enabled.

Table 6-19. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	events	Struct containing flags of events to enable or configure

Returns Status of the events setup procedure.

Table 6-20. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_INVALID_ARG	An invalid configuration option or argument was supplied

6.4.2.2 Function tcc_disable_events()

Disables the event input or output of a TCC instance.

```
void tcc_disable_events(
  struct tcc_module *const module_inst,
  struct tcc_events *const events)
```

Disables one or more input or output events for the given TCC module. See tcc_events for a list of events this module supports.

Note Events cannot be altered while the module is enabled.

Table 6-21. 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

6.4.3 Enable/Disable/Reset

6.4.3.1 Function tcc_enable()

Enable the TCC module.

```
void tcc_enable(
  const struct tcc_module *const module_inst)
```

Enables a TCC module that has been previously initialized. The counter will start when the counter is enabled.

Note When the counter is configured to re-trigger on an event, the counter will not start until the next incoming event appears. Then it restarts on any following event.

Table 6-22. Parameters

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

6.4.3.2 Function tcc_disable()

Disables the TCC module.

```
void tcc_disable(
  const struct tcc_module *const module_inst)
```

Disables a TCC module and stops the counter.

Table 6-23. Parameters

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

6.4.3.3 Function tcc_reset()

Resets the TCC module.

```
void tcc_reset(
  const struct tcc_module *const module_inst)
```

Resets the TCC module, restoring all hardware module registers to their default values and disabling the module. The TCC module will not be accessible while the reset is being performed.

Note When resetting a 32-bit counter only the master TCC module's instance structure should be passed to the function.

Table 6-24. Parameters

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

6.4.4 Set/Toggle Count Direction

6.4.4.1 Function tcc_set_count_direction()

Sets the TCC module count direction.

```
void tcc_set_count_direction(
  const struct tcc_module *const module_inst,
  enum tcc_count_direction dir)
```

Sets the count direction of an initialized TCC module. The specified TCC module can remain running or stopped.

Table 6-25. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	dir	New timer count direction to set

6.4.4.2 Function tcc_toggle_count_direction()

Toggles the TCC module count direction.

```
void tcc_toggle_count_direction(
  const struct tcc_module *const module_inst)
```

Toggles the count direction of an initialized TCC module. The specified TCC module can remain running or stopped.

Table 6-26. Parameters

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

6.4.5 Get/Set Count Value

6.4.5.1 Function tcc_get_count_value()

Get count value of the given TCC module.

```
uint32_t tcc_get_count_value(
  const struct tcc_module *const module_inst)
```

Retrieves the current count value of a TCC module. The specified TCC module can remain running or stopped.

Table 6-27. Parameters

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

Returns Count value of the specified TCC module.

6.4.5.2 Function tcc_set_count_value()

Sets count value for the given TCC module.

```
enum status_code tcc_set_count_value(
  const struct tcc_module *const module_inst,
  const uint32_t count)
```

Sets the timer count value of an initialized TCC module. The specified TCC module can remain running or stopped.

Table 6-28. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	count	New timer count value to set

Returns Status which indicates whether the new value is set.

Table 6-29. Return Values

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

6.4.6 Stop/Restart Counter

6.4.6.1 Function tcc_stop_counter()

Stops the counter.

```
void tcc_stop_counter(
  const struct tcc_module *const module_inst)
```

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

Table 6-30. Parameters

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

6.4.6.2 Function tcc_restart_counter()

Starts the counter from beginning.

```
void tcc_restart_counter(
  const struct tcc_module *const module_inst)
```

Restarts an initialized TCC module's counter.

Table 6-31. Parameters

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

6.4.7 Get/Set Compare/Capture Register

6.4.7.1 Function tcc_get_capture_value()

Gets the TCC module capture value.

```
uint32_t tcc_get_capture_value(
  const struct tcc_module *const module_inst,
  const enum tcc_match_capture_channel channel_index)
```

Retrieves the capture value in the indicated TCC module capture channel.

Table 6-32. 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.

6.4.7.2 Function tcc_set_compare_value()

Sets a TCC module compare value.

```
enum status_code tcc_set_compare_value(
  const struct tcc_module *const module_inst,
  const enum tcc_match_capture_channel channel_index,
  const uint32_t compare)
```

Writes a compare value to the given TCC module compare/capture channel.

If double buffering is enabled it always write to the buffer register. The value will then be updated immediately by calling tcc_force_double_buffer_update(), or be updated when the lock update bit is cleared and the UPDATE condition happen.

Table 6-33. 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 6-34. Return Values

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

6.4.8 Set Top Value

6.4.8.1 Function tcc_set_top_value()

Set the timer TOP/PERIOD value.

```
enum status_code tcc_set_top_value(
  const struct tcc_module *const module_inst,
  const uint32_t top_value)
```

This function writes the given value to the PER/PERB register.

If double buffering is enabled it always write to the buffer register (PERB). The value will then be updated immediately by calling tcc_force_double_buffer_update(), or be updated when the lock update bit is cleared and the UPDATE condition happen.

When using MFRQ, the top value is defined by the CC0 register value and the PER value is ignored, so tcc_set_compare_value (module,channel_0,value) must be used instead of this function to change the actual top value in that case. For all other waveforms operation the top value is defined by PER register value.

Table 6-35. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	top_value	New value to be loaded into the PER/PERB register

Returns Status of the TOP set procedure.

Table 6-36. Return Values

Return value	Description
STATUS_OK	The timer TOP value was updated successfully
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied or top/period value exceed resolution

6.4.9 Set Output Pattern

6.4.9.1 Function tcc_set_pattern()

Sets the TCC module waveform output pattern.

```
enum status_code tcc_set_pattern(
  const struct tcc_module *const module_inst,
  const uint32_t line_index,
  const enum tcc_output_pattern pattern)
```

Force waveform output line to generate specific pattern (0, 1, or as is).

If double buffering is enabled it always write to the buffer register. The value will then be updated immediately by calling tcc_force_double_buffer_update(), or be updated when the lock update bit is cleared and the UPDATE condition happen.

Table 6-37. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	line_index	Output line index
[in]	pattern	Output pattern to use (tcc_output_pattern)

Returns Status of the pattern set procedure.

Table 6-38. Return Values

Return value	Description
STATUS_OK	The PATT register is updated successfully

Return value	Description
STATUS_ERR_INVALID_ARG	An invalid line index was supplied

6.4.10 Set Ramp Index

6.4.10.1 Function tcc_set_ramp_index()

Sets the TCC module ramp index on next cycle.

```
void tcc_set_ramp_index(
  const struct tcc_module *const module_inst,
  const enum tcc_ramp_index ramp_index)
```

In RAMP2 and RAMP2A operation, we can force either cycle A or cycle B at the output, on the next clock cycle. When ramp index command is disabled, cycle A and cycle B will appear at the output, on alternate clock cycles. See tcc_ramp.

Table 6-39. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	ramp_index	Ramp index (tcc_ramp_index) of the next cycle

6.4.11 Status Management

6.4.11.1 Function tcc_is_running()

Checks if the timer/counter is running.

```
bool tcc_is_running(
  struct tcc_module *const module_inst)
```

Table 6-40. Parameters

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

Returns Status which indicates whether the module is running.

Table 6-41. Return Values

Return value	Description
true	The timer/counter is running.
false	The timer/counter is stopped.

6.4.11.2 Function tcc_get_status()

Retrieves the current module status.

```
uint32_t tcc_get_status(
  struct tcc_module *const module_inst)
```

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

Table 6-42. Parameters

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

Returns

Bitmask of TCC_STATUS_* flags.

Table 6-43. Return Values

Return value	Description
TCC_STATUS_CHANNEL_MATCH_CAPTURE(n)	Channel n match/capture has occured
TCC_STATUS_CHANNEL_OUTPUT(n)	Channel n match/capture output state
TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR	Non-recoverable fault x has occured
TCC_STATUS_RECOVERABLE_FAULT_OCCUR(n)	Recoverable fault n has occured
TCC_STATUS_NON_RECOVERABLE_FAULT_PRESE	Non-recoverable fault x input present
TCC_STATUS_RECOVERABLE_FAULT_PRESENT(n)	Recoverable fault n input present
TCC_STATUS_SYNC_READY	None of register is syncing
TCC_STATUS_CAPTURE_OVERFLOW	Timer capture data has overflowed
TCC_STATUS_COUNTER_EVENT	Timer counter event has occurred
TCC_STATUS_COUNT_OVERFLOW	Timer count value has overflowed
TCC_STATUS_COUNTER_RETRIGGERED	Timer counter has been retriggered
TCC_STATUS_STOP	Timer counter has been stopped
TCC_STATUS_RAMP_CYCLE_INDEX	Wave ramp index for cycle

6.4.11.3 Function tcc_clear_status()

Clears a module status flag.

```
void tcc_clear_status(
  struct tcc_module *const module_inst,
  const uint32_t status_flags)
```

Clears the given status flag of the module.

Table 6-44. Parameters

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

6.4.12 Double Buffering Management

6.4.12.1 Function tcc_enable_double_buffering()

Enable TCC double buffering write.

```
void tcc_enable_double_buffering(
  struct tcc_module *const module_inst)
```

When double buffering write is enabled, following function will write values to buffered registers instead of effective ones (buffered):

- PERB: through tcc_set_top_value()
- CCBx(x is 0~3): through tcc_set_compare_value()
- PATTB: through tcc_set_pattern()

Then on UPDATE condition the buffered registers are committed to regular ones to take effect.

Table 6-45. Parameters

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

6.4.12.2 Function tcc_disable_double_buffering()

Disable TCC double buffering Write.

```
void tcc_disable_double_buffering(
  struct tcc_module *const module_inst)
```

When double buffering write is disabled, following function will write values to effective registers (not buffered):

- PER: through tcc_set_top_value()
- CCx(x is 0~3): through tcc_set_compare_value()
- PATT: through tcc_set_pattern()

Note This function does not lock double buffer update, which means on next UPDATE condition the last written buffered values will be committed to take effect. Invoke tcc_lock_double_buffer_update() before this function to disable double buffering update, if this change is not expected.

Table 6-46. Parameters

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

6.4.12.3 Function tcc_lock_double_buffer_update()

Lock the TCC double buffered registers updates.

```
void tcc_lock_double_buffer_update(
  struct tcc_module *const module_inst)
```

Locks the double buffered registers so they will not be updated through their buffered values on UPDATE conditions.

Table 6-47. Parameters

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

6.4.12.4 Function tcc_unlock_double_buffer_update()

Unlock the TCC double buffered registers updates.

```
void tcc_unlock_double_buffer_update(
  struct tcc_module *const module_inst)
```

Unlock the double buffered registers so they will be updated through their buffered values on UPDATE conditions.

Table 6-48. Parameters

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

6.4.12.5 Function tcc_force_double_buffer_update()

Force the TCC double buffered registers to update once.

```
void tcc_force_double_buffer_update(
  struct tcc_module *const module_inst)
```

Table 6-49. Parameters

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

6.4.12.6 Function tcc_enable_circular_buffer_top()

Enable Circular option for double buffered Top/Period Values.

```
void tcc_enable_circular_buffer_top(
  struct tcc_module *const module_inst)
```

Enable circular option for the double buffered top/period values. On each UPDATE condition, the contents of PERB and PER are switched, meaning that the contents of PERB are transferred to PER and the contents of PER are transferred to PERB.

Table 6-50. Parameters

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

6.4.12.7 Function tcc_disable_circular_buffer_top()

Disable Circular option for double buffered Top/Period Values.

```
void tcc_disable_circular_buffer_top(
  struct tcc_module *const module_inst)
```

Stop circularing the double buffered top/period values.

Table 6-51. Parameters

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

6.4.12.8 Function tcc_set_double_buffer_top_values()

Set the timer TOP/PERIOD value and buffer value.

```
enum status_code tcc_set_double_buffer_top_values(
  const struct tcc_module *const module_inst,
  const uint32_t top_value,
  const uint32_t top_buffer_value)
```

This function writes the given value to the PER and PERB register. Usually as preparation for double buffer or circulared double buffer (circular buffer).

When using MFRQ, the top values are defined by the CC0 and CCB0, the PER and PERB values are ignored, so tcc_set_double_buffer_compare_values (module,channel_0,value,buffer) must be used instead of this function to change the actual top values in that case. For all other waveforms operation the top values are defined by PER and PERB registers values.

Table 6-52. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the software module instance struct
[in]	top_value	New value to be loaded into the PER register
[in]	top_buffer_value	New value to be loaded into the PERB register

Returns Status of the TOP set procedure.

Table 6-53. Return Values

Return value	Description
STATUS_OK	The timer TOP value was updated successfully

Return value	Description
STATUS_ERR_INVALID_ARG	An invalid channel index was supplied or top/period value exceed resolution

6.4.12.9 Function tcc_enable_circular_buffer_compare()

Enable Circular option for double buffered Compare Values.

```
enum status_code tcc_enable_circular_buffer_compare(
  struct tcc_module *const module_inst,
  enum tcc_match_capture_channel channel_index)
```

Enable circular option for the double buffered channel compare values. On each UPDATE condition, the contents of CCBx and CCx are switched, meaning that the contents of CCBx are transferred to CCx and the contents of CCx are transferred to CCBx.

Table 6-54. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct
[in]	channel_index	Index of the compare channel to set up to

Table 6-55. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_INVALID_ARG	An invalid channel index is supplied

6.4.12.10 Function tcc_disable_circular_buffer_compare()

Disable Circular option for double buffered Compare Values.

```
enum status_code tcc_disable_circular_buffer_compare(
  struct tcc_module *const module_inst,
  enum tcc_match_capture_channel channel_index)
```

Stop circularing the double buffered compare values.

Table 6-56. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the TCC software instance struct
[in]	channel_index	Index of the compare channel to set up to

Table 6-57. Return Values

Return value	Description
STATUS_OK	The module was initialized successfully
STATUS_INVALID_ARG	An invalid channel index is supplied

6.4.12.11 Function tcc_set_double_buffer_compare_values()

Sets a TCC module compare value and buffer value.

```
enum status_code tcc_set_double_buffer_compare_values(
  struct tcc_module *const module_inst,
  enum tcc_match_capture_channel channel_index,
  const uint32_t compare,
  const uint32_t compare_buffer)
```

Writes compare value and buffer to the given TCC module compare/capture channel. Usually as preparation for double buffer or circulared double buffer (circular buffer).

Table 6-58. Parameters

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

Returns Status of the compare update procedure.

Table 6-59. Return Values

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

6.5 Enumeration Definitions

6.5.1 Enum tcc_callback

Enum for the possible callback types for the TCC module.

Table 6-60. Members

Enum value	Description
TCC_CALLBACK_OVERFLOW	Callback for TCC overflow.
TCC_CALLBACK_RETRIGGER	Callback for TCC Retrigger.
TCC_CALLBACK_COUNTER_EVENT	Callback for TCC counter event.
TCC_CALLBACK_ERROR	Callback for capture overflow error.
TCC_CALLBACK_FAULTA	Callback for Recoverable Fault A.
TCC_CALLBACK_FAULTB	Callback for Recoverable Fault B.
TCC_CALLBACK_FAULT0	Callback for Non-Recoverable Fault. 0.
TCC_CALLBACK_FAULT1	Callback for Non-Recoverable Fault. 1.

Enum value	Description
TCC_CALLBACK_CHANNEL_n	Channel callback type table for TCC Each TCC module may contain several callback types for channels; each channel will have its own callback type in the table, with the channel index number substituted for "n" in the channel callback type (e.g. TCC_MATCH_CAPTURE_CHANNEL_0).

6.5.2 Enum tcc_channel_function

To set a timer channel either in compare or in capture mode.

Table 6-61. Members

Enum value	Description
TCC_CHANNEL_FUNCTION_COMPARE	TCC channel performs compare operation.
TCC_CHANNEL_FUNCTION_CAPTURE	TCC channel performs capture operation.

6.5.3 Enum tcc_clock_prescaler

This enum is used to choose the clock prescaler configuration. The prescaler divides the clock frequency of the TCC module to operate TCC at a slower clock rate.

Table 6-62. Members

Enum value	Description
TCC_CLOCK_PRESCALER_DIV1	Divide clock by 1.
TCC_CLOCK_PRESCALER_DIV2	Divide clock by 2.
TCC_CLOCK_PRESCALER_DIV4	Divide clock by 4.
TCC_CLOCK_PRESCALER_DIV8	Divide clock by 8.
TCC_CLOCK_PRESCALER_DIV16	Divide clock by 16.
TCC_CLOCK_PRESCALER_DIV64	Divide clock by 64.
TCC_CLOCK_PRESCALER_DIV256	Divide clock by 256.
TCC_CLOCK_PRESCALER_DIV1024	Divide clock by 1024.

6.5.4 Enum tcc_count_direction

Used when selecting the Timer/Counter count direction.

Table 6-63. Members

Enum value	Description
TCC_COUNT_DIRECTION_UP	Timer should count upward.
TCC_COUNT_DIRECTION_DOWN	Timer should count downward.

6.5.5 Enum tcc_event0_action

Event action to perform when the module is triggered by event0.

Table 6-64. Members

Enum value	Description
TCC_EVENT0_ACTION_OFF	No event action.
TCC_EVENT0_ACTION_RETRIGGER	Re-trigger Counter on event.
TCC_EVENT0_ACTION_COUNT_EVENT	Count events (increment or decrement, depending on count direction).
TCC_EVENT0_ACTION_START	Start counter on event.
TCC_EVENT0_ACTION_INCREMENT	Increment counter on event.
TCC_EVENT0_ACTION_COUNT_DURING_ACTIVE	Count during active state of asynchronous event.
TCC_EVENT0_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event.

6.5.6 Enum tcc_event1_action

Event action to perform when the module is triggered by event1.

Table 6-65. Members

Enum value	Description
TCC_EVENT1_ACTION_OFF	No event action.
TCC_EVENT1_ACTION_RETRIGGER	Re-trigger Counter on event.
TCC_EVENT1_ACTION_DIR_CONTROL	The event source must be an asynchronous event, input value will override the direction settings. If TCEINVx is 0 and input event is LOW: counter will count up. If TCEINVx is 0 and input event is HIGH: counter will count down.
TCC_EVENT1_ACTION_STOP	Stop counter on event.
TCC_EVENT1_ACTION_DECREMENT	Decrement on event.
TCC_EVENT1_ACTION_PERIOD_PULSE_WIDTH_CAPTURI	Store period in capture register 0, pulse width in capture register 1.
TCC_EVENT1_ACTION_PULSE_WIDTH_PERIOD_CAPTURI	Store pulse width in capture register 0, period in capture register 1.
TCC_EVENT1_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event.

6.5.7 Enum tcc_event_action

Event action to perform when the module is triggered by events.

Table 6-66. Members

Enum value	Description
TCC_EVENT_ACTION_OFF	No event action.
TCC_EVENT_ACTION_STOP	Stop counting, the counter will maintain its current value, waveforms are set to a defined Non-Recoverable State output (tcc_non_recoverable_state_output).
TCC_EVENT_ACTION_RETRIGGER	Re-trigger counter on event, may generate an event if the re-trigger event output is enabled.

Enum value	Description
Note	When re-trigger event action is enabled, enabling the counter will not start until the next incoming event appears.
TCC_EVENT_ACTION_START	Start counter when previously stopped. Start counting on the event rising edge. Further events will not restart the counter; the counter keeps on counting using prescaled GCLK_TCCx, until it reaches TOP or Zero depending on the direction.
TCC_EVENT_ACTION_COUNT_EVENT	Count events; i.e. Increment or decrement depending on count direction.
TCC_EVENT_ACTION_DIR_CONTROL	The event source must be an asynchronous event, input value will overrides the direction settings (input low: counting up, input high counting down).
TCC_EVENT_ACTION_INCREMENT	Increment the counter on event, irrespective of count direction.
TCC_EVENT_ACTION_DECREMENT	Decrement the counter on event, irrespective of count direction.
TCC_EVENT_ACTION_COUNT_DURING_ACTIVE	Count during active state of asynchronous event. In this case, depending on the count direction, the count will be incremented or decremented on each prescaled GCLK_TCCx, as long as the input event remains active.
TCC_EVENT_ACTION_PERIOD_PULSE_WIDTH_CAPTURE	Store period in capture register 0, pulse width in capture register 1.
TCC_EVENT_ACTION_PULSE_WIDTH_PERIOD_CAPTURE	Store pulse width in capture register 0, period in capture register 1.
TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT	Generate Non-Recoverable Fault on event.

6.5.8 Enum tcc_event_generation_selection

This enum is used to define the point at which the counter event is generated.

Table 6-67. Members

Enum value	Description
TCC_EVENT_GENERATION_SELECTION_START	Counter Event is generated when a new counter cycle starts.
TCC_EVENT_GENERATION_SELECTION_END	Counter Event is generated when a counter cycle ends.
TCC_EVENT_GENERATION_SELECTION_BETWEEN	Counter Event is generated when a counter cycle ends, except for the first and last cycles.
TCC_EVENT_GENERATION_SELECTION_BOUNDARY	Counter Event is generated when a new counter cycle starts or ends.

6.5.9 Enum tcc_fault_blanking

Table 6-68. Members

Enum value	Description
TCC_FAULT_BLANKING_DISABLE	No blanking.
TCC_FAULT_BLANKING_RISING_EDGE	Blanking applied from rising edge of the output waveform.
TCC_FAULT_BLANKING_FALLING_EDGE	Blanking applied from falling edge of the output waveform.
TCC_FAULT_BLANKING_BOTH_EDGE	Blanking applied from each toggle of the output waveform.

6.5.10 Enum tcc_fault_capture_action

Table 6-69. Members

Enum value	Description
TCC_FAULT_CAPTURE_DISABLE	Capture disabled.
TCC_FAULT_CAPTURE_EACH	Capture on Fault, each value is captured.
TCC_FAULT_CAPTURE_MINIMUM	Capture the minimum detection, but notify on smaller ones.
TCC_FAULT_CAPTURE_MAXIMUM	Capture the maximum detection, but notify on bigger ones.
TCC_FAULT_CAPTURE_SMALLER	Capture if the value is smaller than last, notify event or interrupt if previous stamp is confirmed to be "local minimum" (not bigger than current stamp).
TCC_FAULT_CAPTURE_BIGGER	Capture if the value is bigger than last, notify event or interrupt if previous stamp is confirmed to be "local maximum" (not smaller than current stamp).
TCC_FAULT_CAPTURE_CHANGE	Capture if the time stamps changes its increment direction.

6.5.11 Enum tcc_fault_capture_channel

Table 6-70. Members

Enum value	Description
TCC_FAULT_CAPTURE_CHANNEL_0	Recoverable fault triggers channel 0 capture operation.
TCC_FAULT_CAPTURE_CHANNEL_1	Recoverable fault triggers channel 1 capture operation.
TCC_FAULT_CAPTURE_CHANNEL_2	Recoverable fault triggers channel 2 capture operation.
TCC_FAULT_CAPTURE_CHANNEL_3	Recoverable fault triggers channel 3 capture operation.

6.5.12 Enum tcc_fault_halt_action

Table 6-71. Members

Enum value	Description
TCC_FAULT_HALT_ACTION_DISABLE	Halt action disabled.
TCC_FAULT_HALT_ACTION_HW_HALT	Hardware halt action, counter is halted until restart.
TCC_FAULT_HALT_ACTION_SW_HALT	Software halt action, counter is halted until fault bit cleared.
TCC_FAULT_HALT_ACTION_NON_RECOVERABLE	Non-Recoverable fault, force output to pre-defined level.

6.5.13 Enum tcc_fault_keep

Table 6-72. Members

Enum value	Description
TCC_FAULT_KEEP_DISABLE	Disable keeping, wave output released as soon as fault is released.
TCC_FAULT_KEEP_TILL_END	Keep wave output until end of TCC cycle.

6.5.14 Enum tcc_fault_qualification

Table 6-73. Members

Enum value	Description
TCC_FAULT_QUALIFICATION_DISABLE	The input is not disabled on compare condition.
TCC_FAULT_QUALIFICATION_BY_OUTPUT	The input is disabled when match output signal is at inactive level.

6.5.15 Enum tcc_fault_restart

Table 6-74. Members

Enum value	Description
TCC_FAULT_RESTART_DISABLE	Restart Action disabled.
TCC_FAULT_RESTART_ENABLE	Restart Action enabled.

6.5.16 Enum tcc_fault_source

Table 6-75. Members

Enum value	Description
TCC_FAULT_SOURCE_DISABLE	Fault input is disabled.
TCC_FAULT_SOURCE_ENABLE	Match Capture Event x (x=0,1) input.
TCC_FAULT_SOURCE_INVERT	Inverted MCEx (x=0,1) event input.
TCC_FAULT_SOURCE_ALTFAULT	Alternate fault (A or B) state at the end of the previous period.

6.5.17 Enum tcc_fault_state_output

Table 6-76. Members

Enum value	Description
TCC_FAULT_STATE_OUTPUT_OFF	Non-recoverable fault output is tri-stated.
TCC_FAULT_STATE_OUTPUT_0	Non-recoverable fault force output 0.
TCC_FAULT_STATE_OUTPUT_1	Non-recoverable fault force output 1.

6.5.18 Enum tcc_match_capture_channel

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

Table 6-77. Members

Enum value	Description
TCC_MATCH_CAPTURE_CHANNEL_n	Match capture channel index table for TCC Each TCC module may contain several match capture channels; each channel will have its own index in the table, with the index number substituted for "n" in the index name (e.g. TCC_MATCH_CAPTURE_CHANNEL_0).

6.5.19 Enum tcc_output_invertion

Used when enabling or disabling output inversion.

Table 6-78. Members

Enum value	Description
TCC_OUTPUT_INVERTION_DISABLE	Output inversion not to be enabled.
TCC_OUTPUT_INVERTION_ENABLE	Invert the output from WO[x].

6.5.20 Enum tcc_output_pattern

Used when disabling output pattern or when selecting a specific pattern.

Table 6-79. Members

Enum value	Description
TCC_OUTPUT_PATTERN_DISABLE	SWAP Output pattern is not used
TCC_OUTPUT_PATTERN_0	Pattern 0 is applied to SWAP output
TCC_OUTPUT_PATTERN_1	Pattern 1 is applied to SWAP output

6.5.21 Enum tcc_ramp

Ramp Operations which are supported in single-slope PWM generation.

Table 6-80. Members

Enum value	Description
TCC_RAMP_RAMP1	Default timer/counter PWM operation.

Enum value	Description
TCC_RAMP_RAMP2A	Uses a single channel (CC0) to control both CC0/CC1 compare outputs. In cycle A, the channel 0 output is disabled, and in cycle B, the channel 1 output is disabled.
TCC_RAMP_RAMP2	Uses channels CC0 and CC1 to control compare outputs. In cycle A, the channel 0 output is disabled, and in cycle B, the channel 1 output is disabled.

6.5.22 Enum tcc_ramp_index

In ramp operation, each two period cycles are marked as cycle A and B, the index 0 represents cycle A and 1 represents cycle B.

Table 6-81. Members

Enum value	Description
TCC_RAMP_INDEX_DEFAULT	Default, cycle index toggles.
TCC_RAMP_INDEX_FORCE_B	Force next cycle to be cycle B (set to 1).
TCC_RAMP_INDEX_FORCE_A	Force next cycle to be cycle A (clear to 0).
TCC_RAMP_INDEX_FORCE_KEEP	Force next cycle keeping the same as current.

6.5.23 Enum tcc_reload_action

This enum specify how the counter is reloaded and whether the prescaler should be restarted.

Table 6-82. Members

Enum value	Description
TCC_RELOAD_ACTION_GCLK	The counter is reloaded/reset on the next GCLK and starts counting on the prescaler clock.
TCC_RELOAD_ACTION_PRESC	The counter is reloaded/reset on the next prescaler clock.
TCC_RELOAD_ACTION_RESYNC	The counter is reloaded/reset on the next GCLK, and the prescaler is restarted as well.

6.5.24 Enum tcc_wave_generation

This enum is used to specify the waveform generation mode.

Table 6-83. Members

Enum value	Description
TCC_WAVE_GENERATION_NORMAL_FREQ	Normal Frequency: Top is the PER register, output toggled on each compare match.
TCC_WAVE_GENERATION_MATCH_FREQ	Match Frequency: Top is CC0 register, output toggles on each update condition.
TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM	Single-Slope PWM: Top is the PER register, CCx controls duty cycle (output active when count is greater than CCx).

Enum value	Description
TCC_WAVE_GENERATION_DOUBLE_SLOPE_CRITICAL	Double-slope (count up and down), non centre-aligned: Top is the PER register, CC[x] controls duty cycle while counting up and CC[x+N/2] controls it while counting down.
TCC_WAVE_GENERATION_DOUBLE_SLOPE_BOTTOM	Double-slope (count up and down), interrupt/event at Bottom (Top is the PER register, output active when count is greater than CCx).
TCC_WAVE_GENERATION_DOUBLE_SLOPE_BOTH	Double-slope (count up and down), interrupt/event at Bottom and Top: (Top is the PER register, output active when count is lower than CCx).
TCC_WAVE_GENERATION_DOUBLE_SLOPE_TOP	Double-slope (count up and down), interrupt/event at Top (Top is the PER register, output active when count is greater than CCx).

6.5.25 Enum tcc_wave_output

This enum is used to specify which wave output to do operations on.

Table 6-84. Members

Enum value	Description
TCC_WAVE_OUTPUT_n	Waveform output index table for TCC Each TCC module may contain several wave outputs; each output will have its own index in the table, with the index number substituted for "n" in the index name (e.g. TCC_WAVE_OUTPUT_0).

6.5.26 Enum tcc_wave_polarity

Specifies whether the wave output needs to be inverted or not.

Table 6-85. Members

Enum value	Description
TCC_WAVE_POLARITY_0	Wave output is not inverted
TCC_WAVE_POLARITY_1	Wave output is inverted

7. Extra Information for TCC Driver

7.1 Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
DMA	Direct Memory Access
TCC	Timer Counter for Control Applications
PWM	Pulse Width Modulation
PWP	Pulse Width Period
PPW	Period Pulse Width

7.2 Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

7.3 Errata

There are no errata related to this driver.

7.4 Module History

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

Changelog
Add double buffering functionality
Add fault handling functionality
Initial Release

8. Examples for TCC Driver

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

- Quick Start Guide for TCC - Basic
- Quick Start Guide for TCC - Double Buffering and Circular
- Quick Start Guide for TCC - Timer
- Quick Start Guide for TCC - Callback
- Quick Start Guide for TCC - Non-Recoverable Fault
- Quick Start Guide for TCC - Recoverable Fault
- Quick Start Guide for Using DMA with TCC

8.1 Quick Start Guide for TCC - Basic

The supported board list:

- SAM D21/R21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal. Here the pulse width is set to one quarter of the period. When connect PWM output to LED it makes the LED light. To see the waveform, you may need an oscilloscope.

The PWM output is set up as follows:

Board	Pin	Connect to
SAMD21 Xpro	PB30	LED0
SAMR21 Xpro	PA19	LED0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No fault or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward

- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Counter top set to 0xFFFF
- Capture compare channel 0 set to 0xFFFF/4

8.1.1 Quick Start

8.1.1.1 Prerequisites

There are no prerequisites for this use case.

8.1.1.2 Code

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

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

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

```
struct tcc_module tcc_instance;
```

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

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = (0xFFFF / 4);

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    tcc_enable(&tcc_instance);
}
```

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

```
configure_tcc();
```

8.1.1.3 Workflow

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

```
struct tcc_module tcc_instance;
```

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

2. Configure the TCC module.

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

```
struct tcc_config config_tcc;
```

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

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

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

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

```
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = (0xFFFF / 4);
```

   d. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;
```

   e. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

   f. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

8.1.2 Use Case

8.1.2.1 Code

Copy-paste the following code to your user application:

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

8.1.2.2 Workflow

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

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

8.2 Quick Start Guide for TCC - Double Buffering and Circular

The supported board list:

- SAM D21/R21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal. Here the pulse width alters in one quarter and three quarter of the period. When connect PWM output to LED it makes the LED light. To see the waveform, you may need an oscilloscope.

The PWM output is set up as follows:

Board	Pin	Connect to
SAMD21 Xpro	PB30	LED0
SAMR21 Xpro	PA19	LED0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- Prescaler is set to 1024
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No fault or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action

- No event generation enabled
- Counter starts on 0
- Counter top set to 8000
- Capture compare channel set to 8000/4
- Capture compare channel buffer set to 8000*3/4
- Circular option for compare channel is enabled so that the compare values keep switching on update condition

8.2.1 Quick Start

8.2.1.1 Prerequisites

There are no prerequisites for this use case.

8.2.1.2 Code

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

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

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

```
struct tcc_module tcc_instance;
```

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

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV1024;
    config_tcc.counter.period = 8000;
    config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = (8000 / 4);

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    tcc_set_compare_value(&tcc_instance,
            (enum tcc_match_capture_channel)CONF_PWM_CHANNEL, 8000*3/4);
    tcc_enable_circular_buffer_compare(&tcc_instance,
            (enum tcc_match_capture_channel)CONF_PWM_CHANNEL);

    tcc_enable(&tcc_instance);
```

```
}
```

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

```
configure_tcc();
```

8.2.1.3 Workflow

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

```
struct tcc_module tcc_instance;
```

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

2. Configure the TCC module.

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

```
struct tcc_config config_tcc;
```

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

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

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

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

```
config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV1024;
config_tcc.counter.period = 8000;
config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = (8000 / 4);
```

   d. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;
```

   e. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

   f. Set to compare buffer value and enable circular of double buffered compare values.

```
tcc_set_compare_value(&tcc_instance,
```

```
        (enum tcc_match_capture_channel)CONF_PWM_CHANNEL, 8000*3/4);
tcc_enable_circular_buffer_compare(&tcc_instance,
        (enum tcc_match_capture_channel)CONF_PWM_CHANNEL);
```

g. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

8.2.2 Use Case

8.2.2.1 Code

Copy-paste the following code to your user application:

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

8.2.2.2 Workflow

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

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

8.3 Quick Start Guide for TCC - Timer

The supported board list:

- SAM D21/R21 Xplained Pro
- SAM D11 Xplained Pro

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

The TCC module will be set up as follows:

- GCLK generator 1 (GCLK 32K) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- Prescaler is divided by 64
- GCLK reload action
- Count upward
- Don't run in standby
- No waveform outputs
- No capture enabled
- Don't perform one-shot operations
- No event input enabled

- No event action
- No event generation enabled
- Counter starts on 0
- Counter top set to 2000 (about 4s) and generate overflow callback
- Channel 0 is set to compare and match value 900 and generate callback
- Channel 1 is set to compare and match value 930 and generate callback
- Channel 2 is set to compare and match value 1100 and generate callback
- Channel 3 is set to compare and match value 1250 and generate callback

8.3.1 Quick Start

8.3.1.1 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.

8.3.1.2 Code

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

```
struct tcc_module tcc_instance;
```

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

```
static void tcc_callback_to_toggle_led(
        struct tcc_module *const module_inst)
{
    port_pin_toggle_output_level(LED0_PIN);
}
```

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

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, TCC0);

    config_tcc.counter.clock_source = GCLK_GENERATOR_1;
    config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV64;
    config_tcc.counter.period =   2000;
    config_tcc.compare.match[0] =  900;
    config_tcc.compare.match[1] =  930;
    config_tcc.compare.match[2] = 1100;
    config_tcc.compare.match[3] = 1250;

    tcc_init(&tcc_instance, TCC0, &config_tcc);

    tcc_enable(&tcc_instance);
}

static void configure_tcc_callbacks(void)
{
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
```

```
            TCC_CALLBACK_OVERFLOW);
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
            TCC_CALLBACK_CHANNEL_0);
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
            TCC_CALLBACK_CHANNEL_1);
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
            TCC_CALLBACK_CHANNEL_2);
    tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
            TCC_CALLBACK_CHANNEL_3);

    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_OVERFLOW);
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_0);
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_1);
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_2);
    tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_3);
}
```

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

```
configure_tcc();
configure_tcc_callbacks();
```

8.3.1.3 Workflow

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

   ```
   struct tcc_module tcc_instance;
   ```

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

2. Configure the TCC module.

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

      ```
      struct tcc_config config_tcc;
      ```

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

      ```
      tcc_get_config_defaults(&config_tcc, TCC0);
      ```

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

   c. Alter the TCC settings to configure the GCLK source, prescaler, period and compare channel values.

      ```
      config_tcc.counter.clock_source = GCLK_GENERATOR_1;
      config_tcc.counter.clock_prescaler = TCC_CLOCK_PRESCALER_DIV64;
      config_tcc.counter.period =   2000;
      config_tcc.compare.match[0] =  900;
      config_tcc.compare.match[1] =  930;
      config_tcc.compare.match[2] = 1100;
      ```

```
config_tcc.compare.match[3] = 1250;
```

d. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, TCC0, &config_tcc);
```

e. Enable the TCC module to start the timer.

```
tcc_enable(&tcc_instance);
```

3. Configure the TCC callbacks.

a. Register the Overflow and Compare Channel Match callback functions with the driver.

```
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
        TCC_CALLBACK_OVERFLOW);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
        TCC_CALLBACK_CHANNEL_0);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
        TCC_CALLBACK_CHANNEL_1);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
        TCC_CALLBACK_CHANNEL_2);
tcc_register_callback(&tcc_instance, tcc_callback_to_toggle_led,
        TCC_CALLBACK_CHANNEL_3);
```

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

```
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_OVERFLOW);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_0);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_1);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_2);
tcc_enable_callback(&tcc_instance, TCC_CALLBACK_CHANNEL_3);
```

8.3.2 Use Case

8.3.2.1 Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

8.3.2.2 Workflow

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

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

8.4 Quick Start Guide for TCC - Callback

The supported board list:

- SAM D21/R21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. When connect PWM output to LED it makes the LED vary its light. To see the waveform, you may need an oscilloscope.

The PWM output is set up as follows:

Board	Pin	Connect to
SAMD21 Xpro	PB30	LED0
SAMR21 Xpro	PA19	LED0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No faults or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input enabled
- No event action
- No event generation enabled
- Counter starts on 0

8.4.1 Quick Start

8.4.1.1 Prerequisites

There are no prerequisites for this use case.

8.4.1.2 Code

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

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
```

```
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

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

```
struct tcc_module tcc_instance;
```

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

```
static void tcc_callback_to_change_duty_cycle(
        struct tcc_module *const module_inst)
{
    static uint32_t delay = 10;
    static uint32_t i = 0;

    if (--delay) {
        return;
    }
    delay = 10;
    i = (i + 0x0800) & 0xFFFF;
    tcc_set_compare_value(module_inst,
            (enum tcc_match_capture_channel)
                    (TCC_MATCH_CAPTURE_CHANNEL_0 + CONF_PWM_CHANNEL),
            i + 1);
}
```

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

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    tcc_enable(&tcc_instance);
}

static void configure_tcc_callbacks(void)
{
    tcc_register_callback(
            &tcc_instance,
            tcc_callback_to_change_duty_cycle,
            (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));

    tcc_enable_callback(&tcc_instance,
            (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
}
```

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

```
configure_tcc();
configure_tcc_callbacks();
```

8.4.1.3 Workflow

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

```
struct tcc_module tcc_instance;
```

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

2. Configure the TCC module.

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

```
struct tcc_config config_tcc;
```

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

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

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

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

```
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
```

   d. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;
```

   e. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

   f. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

3. Configure the TCC callbacks.

a. Register the Compare Channel 0 Match callback functions with the driver.

```
tcc_register_callback(
        &tcc_instance,
        tcc_callback_to_change_duty_cycle,
        (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

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

```
tcc_enable_callback(&tcc_instance,
        (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

8.4.2 Use Case

8.4.2.1 Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

8.4.2.2 Workflow

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

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

8.5 Quick Start Guide for TCC - Non-Recoverable Fault

The supported kit list:

- SAM D21/R21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. There is a non-recoverable faul input which controls PWM output, when this fault is active (low) the PWM output will be forced to be high. When fault is released (input high) the PWM output then will go on.

When connect PWM output to LED it makes the LED vary its light. If fault input is from a button, the LED will be off when the button is down and on when the button is up. To see the PWM waveform, you may need an oscilloscope.

The PWM output and fault input is set up as follows:

Board	Pin	Connect to
SAMD21 Xpro	PB30	LED0
SAMD21 Xpro	PA15	SW0
SAMR21 Xpro	PA19	LED0
SAMR21 Xpro	PA28	SW0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source

- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No waveform extentions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input except TCC event0 enabled
- No event action except TCC event0 acts as Non-Recoverable Fault
- No event generation enabled
- Counter starts on 0

8.5.1 Quick Start

8.5.1.1 Prerequisites

There are no prerequisites for this use case.

8.5.1.2 Code

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

- SAM D21 Xplained Pro.

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

```
#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15
```

```
#define CONF_FAULT_EVENT_USER      EVSYS_ID_USER_TCC0_EV_0
```

- SAM R21 Xplained Pro.

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

```
#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN

#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX

#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE

#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_8

#define CONF_FAULT_EVENT_USER      EVSYS_ID_USER_TCC0_EV_0
```

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

```
#include <string.h>
```

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

```
struct tcc_module tcc_instance;
```

```
struct events_resource event_resource;
```

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

```
static void tcc_callback_to_change_duty_cycle(
        struct tcc_module *const module_inst)
{
    static uint32_t delay = 10;
    static uint32_t i = 0;

    if (--delay) {
        return;
    }
    delay = 10;
    i = (i + 0x0800) & 0xFFFF;
    tcc_set_compare_value(module_inst,
            (enum tcc_match_capture_channel)
                    (TCC_MATCH_CAPTURE_CHANNEL_0 + CONF_PWM_CHANNEL),
            i + 1);
}
```

```
static void eic_callback_to_clear_halt(void)
```

```
{
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
                TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT(0) |
                TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(0));
    }
}
```

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

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
    config_tcc.wave_ext.non_recoverable_fault[0].output = TCC_FAULT_STATE_OUTPUT_1;
    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    struct tcc_events events;
    memset(&events, 0, sizeof(struct tcc_events));

    events.on_input_event_perform_action[0] = true;
    events.input_config[0].modify_action = true;
    events.input_config[0].action = TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT;

    tcc_enable_events(&tcc_instance, &events);

    tcc_enable(&tcc_instance);
}

static void configure_tcc_callbacks(void)
{
    tcc_register_callback(
            &tcc_instance,
            tcc_callback_to_change_duty_cycle,
            (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));

    tcc_enable_callback(&tcc_instance,
            (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
}
```

```
static void configure_eic(void)
{
    struct extint_chan_conf config;
    extint_chan_get_config_defaults(&config);
    config.filter_input_signal = true;
    config.detection_criteria  = EXTINT_DETECT_BOTH;
    config.gpio_pin     = CONF_FAULT_EIC_PIN;
    config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
    extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);

    struct extint_events events;
```

```
    memset(&events, 0, sizeof(struct extint_events));
    events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
    extint_enable_events(&events);

    extint_register_callback(eic_callback_to_clear_halt,
            CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
    extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
            EXTINT_CALLBACK_TYPE_DETECT);
}
```

```
static void configure_event(void)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator = CONF_FAULT_EVENT_GENERATOR;
    config.path      = EVENTS_PATH_ASYNCHRONOUS;

    events_allocate(&event_resource, &config);

    events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
}
```

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

```
configure_tcc();
configure_tcc_callbacks();

configure_eic();
configure_event();
```

8.5.1.3 Workflow

Configure TCC

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

```
struct tcc_module tcc_instance;
```

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

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

```
struct tcc_config config_tcc;
```

3. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

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

4. Alter the TCC settings to configure the counter width, wave generation mode and the compare channel 0 value and fault options. Here the Non-Recoverable Fault output is enabled and set to high level (1).

```
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
```

```
config_tcc.wave_ext.non_recoverable_fault[0].output = TCC_FAULT_STATE_OUTPUT_1;
```

5. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;
```

6. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

7. Create a TCC events configuration struct, which can be filled out to enable/disable events and configure event settings. Reset all fields to zero.

```
struct tcc_events events;
memset(&events, 0, sizeof(struct tcc_events));
```

8. Alter the TCC events settings to enable/disable desired events, to change event generating options and modify event actions. Here TCC event0 will act as Non-Recoverable Fault input.

```
events.on_input_event_perform_action[0] = true;
events.input_config[0].modify_action = true;
events.input_config[0].action = TCC_EVENT_ACTION_NON_RECOVERABLE_FAULT;
```

9. Enable and apply events settings.

```
tcc_enable_events(&tcc_instance, &events);
```

10. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

11. Register the Compare Channel 0 Match callback functions with the driver.

```
tcc_register_callback(
        &tcc_instance,
        tcc_callback_to_change_duty_cycle,
        (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

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

```
tcc_enable_callback(&tcc_instance,
        (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

Configure EXTINT for fault input

1. Create an EXTINT module channel configuration struct, which can be filled out to adjust the configuration of a single external interrupt channel.

```
struct extint_chan_conf config;
```

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

```
extint_chan_get_config_defaults(&config);
```

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

3. Adjust the configuration struct to configure the pin MUX (to route the desired physical pin to the logical channel) to the board button, and to configure the channel to detect both rising and falling edges.

```
config.filter_input_signal = true;
config.detection_criteria  = EXTINT_DETECT_BOTH;
config.gpio_pin     = CONF_FAULT_EIC_PIN;
config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
```

4. Configure external interrupt channel with the desired channel settings.

```
extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);
```

5. Create a TXTINT events configuration struct, which can be filled out to enable/disable events. Reset all fields to zero.

```
struct extint_events events;
memset(&events, 0, sizeof(struct extint_events));
```

6. Adjust the configuration struct, set the channels to be enabled to `true`. Here the channel to the board button is used.

```
events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
```

7. Enable the events.

```
extint_enable_events(&events);
```

8. Define the EXTINT callback that will be fired when a detection event occurs. For this example, when fault line is released, the TCC fault state is cleared to go on PWM generating.

```
static void eic_callback_to_clear_halt(void)
{
```

```
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
                TCC_STATUS_NON_RECOVERABLE_FAULT_PRESENT(0) |
                TCC_STATUS_NON_RECOVERABLE_FAULT_OCCUR(0));
    }
}
```

9. Register a callback function `eic_callback_to_clear_halt()` to handle detections from the External Interrupt Controller (EIC).

```
extint_register_callback(eic_callback_to_clear_halt,
        CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
```

10. Enable the registered callback function for the configured External Interrupt channel, so that it will be called by the module when the channel detects an edge.

```
extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
        EXTINT_CALLBACK_TYPE_DETECT);
```

Configure EVENTS for fault input

1. Create a event resource instance struct for the EVENTS module to store.

```
struct events_resource event_resource;
```

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

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

```
struct events_config config;
```

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

```
events_get_config_defaults(&config);
```

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

4. Adjust the configuration struct to request that the channel be attached to the specified event generator, and that the asynchronous event path be used. Here the EIC channel connected to board button is the event generator.

```
config.generator = CONF_FAULT_EVENT_GENERATOR;
config.path      = EVENTS_PATH_ASYNCHRONOUS;
```

5. Allocate and configure the channel using the configuration structure.

```
events_allocate(&event_resource, &config);
```

Note The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

6. Attach an user to the channel. Here the user is TCC event0, which has been configured as input of Non-Recoverable Fault.

```
events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
```

8.5.2 Use Case

8.5.2.1 Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

8.5.2.2 Workflow

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

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

8.6 Quick Start Guide for TCC - Recoverable Fault

The supported board list:

- SAM D21/R21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal, with a varying duty cycle. Here the pulse width is increased each time the timer count matches the set compare value. There is a recoverable faul input which controls PWM output, when this fault is active (low) the PWM output will be frozen (could be off or on, no light changing). When fault is released (input high) the PWM output then will go on.

When connect PWM output to LED it makes the LED vary its light. If fault input is from a button, the LED will be frozen and not changing it's light when the button is down and will go on when the button is up. To see the PWM waveform, you may need an oscilloscope.

The PWM output and fault input is set up as follows:

Board	Pin	Connect to
SAMD21 Xpro	PB30	LED0
SAMD21 Xpro	PA15	SW0
SAMR21 Xpro	PA06	EXT1 Pin 3
SAMR21 Xpro	PA28	SW0

The TCC module will be set up as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API

- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby
- No waveform extentions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- No event input except channel 0 event enabled
- No event action
- No event generation enabled
- Counter starts on 0
- Recoverable Fault A is generated from channel 0 event input, fault halt acts as software halt, other actions or options are all disabled

8.6.1 Quick Start

8.6.1.1 Prerequisites

There are no prerequisites for this use case.

8.6.1.2 Code

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

- SAM D21 Xplained Pro.

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

```
#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN
#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX
#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE
#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15
```

```
#define CONF_FAULT_EVENT_USER      EVSYS_ID_USER_TCC0_MC_0
```

- SAM R21 Xplained Pro.

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE

#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL

#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT

#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN

#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

```
#define CONF_FAULT_EIC_PIN       SW0_EIC_PIN

#define CONF_FAULT_EIC_PIN_MUX   SW0_EIC_PINMUX

#define CONF_FAULT_EIC_LINE      SW0_EIC_LINE

#define CONF_FAULT_EVENT_GENERATOR EVSYS_ID_GEN_EIC_EXTINT_15

#define CONF_FAULT_EVENT_USER      EVSYS_ID_USER_TCC0_MC_0
```

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

```
#include <string.h>
```

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

```
struct tcc_module tcc_instance;
```

```
struct events_resource event_resource;
```

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

```
static void tcc_callback_to_change_duty_cycle(
        struct tcc_module *const module_inst)
{
    static uint32_t delay = 10;
    static uint32_t i = 0;

    if (--delay) {
        return;
    }
    delay = 10;
    i = (i + 0x0800) & 0xFFFF;
    tcc_set_compare_value(module_inst,
            (enum tcc_match_capture_channel)
                    (TCC_MATCH_CAPTURE_CHANNEL_0 + CONF_PWM_CHANNEL),
            i + 1);
}
```

```
static void eic_callback_to_clear_halt(void)
```

```
{
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
                TCC_STATUS_RECOVERABLE_FAULT_PRESENT(CONF_PWM_CHANNEL) |
                TCC_STATUS_RECOVERABLE_FAULT_OCCUR(CONF_PWM_CHANNEL));
    }
}
```

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

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0xFFFF;
    config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
    config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].source =
            TCC_FAULT_SOURCE_ENABLE;
    config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].halt_action =
            TCC_FAULT_HALT_ACTION_SW_HALT;
    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    struct tcc_events events;
    memset(&events, 0, sizeof(struct tcc_events));

    events.on_event_perform_channel_action[CONF_PWM_CHANNEL] = true;

    tcc_enable_events(&tcc_instance, &events);

    tcc_enable(&tcc_instance);
}

static void configure_tcc_callbacks(void)
{
    tcc_register_callback(
            &tcc_instance,
            tcc_callback_to_change_duty_cycle,
            (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));

    tcc_enable_callback(&tcc_instance,
            (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
}
```

```
static void configure_eic(void)
{
    struct extint_chan_conf config;
    extint_chan_get_config_defaults(&config);
    config.filter_input_signal = true;
    config.detection_criteria  = EXTINT_DETECT_BOTH;
    config.gpio_pin     = CONF_FAULT_EIC_PIN;
    config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
    extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);
```

```
    struct extint_events events;
    memset(&events, 0, sizeof(struct extint_events));
    events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
    extint_enable_events(&events);

    extint_register_callback(eic_callback_to_clear_halt,
            CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
    extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
            EXTINT_CALLBACK_TYPE_DETECT);
}
```

```
static void configure_event(void)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator = CONF_FAULT_EVENT_GENERATOR;
    config.path      = EVENTS_PATH_ASYNCHRONOUS;

    events_allocate(&event_resource, &config);

    events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
}
```

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

```
configure_tcc();
configure_tcc_callbacks();

configure_eic();
configure_event();
```

8.6.1.3 Workflow

Configure TCC

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

```
struct tcc_module tcc_instance;
```

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

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

```
struct tcc_config config_tcc;
```

3. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

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

4. Alter the TCC settings to configure the counter width, wave generation mode and the compare channel 0 value and fault options. Here the Recoverable Fault input is enabled and halt action is set to software mode (must use software to clear halt state).

```
config_tcc.counter.period = 0xFFFF;
config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.match[CONF_PWM_CHANNEL] = 0xFFFF;
```

```
config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].source =
        TCC_FAULT_SOURCE_ENABLE;
config_tcc.wave_ext.recoverable_fault[CONF_PWM_CHANNEL].halt_action =
        TCC_FAULT_HALT_ACTION_SW_HALT;
```

5. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;
```

6. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

7. Create a TCC events configuration struct, which can be filled out to enable/disable events and configure event settings. Reset all fields to zero.

```
struct tcc_events events;
memset(&events, 0, sizeof(struct tcc_events));
```

8. Alter the TCC events settings to enable/disable desired events, to change event generating options and modify event actions. Here channel event 0 input is enabled as source of recoverable fault.

```
events.on_event_perform_channel_action[CONF_PWM_CHANNEL] = true;
```

9. Enable and apply events settings.

```
tcc_enable_events(&tcc_instance, &events);
```

10. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

11. Register the Compare Channel 0 Match callback functions with the driver.

```
tcc_register_callback(
        &tcc_instance,
        tcc_callback_to_change_duty_cycle,
```

```
        (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

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

```
tcc_enable_callback(&tcc_instance,
        (enum tcc_callback)(TCC_CALLBACK_CHANNEL_0 + CONF_PWM_CHANNEL));
```

Configure EXTINT for fault input

1. Create an EXTINT module channel configuration struct, which can be filled out to adjust the configuration of a single external interrupt channel.

```
struct extint_chan_conf config;
```

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

```
extint_chan_get_config_defaults(&config);
```

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

3. Adjust the configuration struct to configure the pin MUX (to route the desired physical pin to the logical channel) to the board button, and to configure the channel to detect both rising and falling edges.

```
config.filter_input_signal = true;
config.detection_criteria  = EXTINT_DETECT_BOTH;
config.gpio_pin     = CONF_FAULT_EIC_PIN;
config.gpio_pin_mux = CONF_FAULT_EIC_PIN_MUX;
```

4. Configure external interrupt channel with the desired channel settings.

```
extint_chan_set_config(CONF_FAULT_EIC_LINE, &config);
```

5. Create a TXTINT events configuration struct, which can be filled out to enable/disable events. Reset all fields to zero.

```
struct extint_events events;
memset(&events, 0, sizeof(struct extint_events));
```

6. Adjust the configuration struct, set the channels to be enabled to `true`. Here the channel to the board button is used.

```
events.generate_event_on_detect[CONF_FAULT_EIC_LINE] = true;
```

7. Enable the events.

```
extint_enable_events(&events);
```

8. Define the EXTINT callback that will be fired when a detection event occurs. For this example, when fault line is released, the TCC fault state is cleared to go on PWM generating.

```
static void eic_callback_to_clear_halt(void)
```

```
{
    if (port_pin_get_input_level(CONF_FAULT_EIC_PIN)) {
        tcc_clear_status(&tcc_instance,
                TCC_STATUS_RECOVERABLE_FAULT_PRESENT(CONF_PWM_CHANNEL) |
                TCC_STATUS_RECOVERABLE_FAULT_OCCUR(CONF_PWM_CHANNEL));
    }
}
```

9. Register a callback function `eic_callback_to_clear_halt()` to handle detections from the External Interrupt Controller (EIC).

```
extint_register_callback(eic_callback_to_clear_halt,
        CONF_FAULT_EIC_LINE, EXTINT_CALLBACK_TYPE_DETECT);
```

10. Enable the registered callback function for the configured External Interrupt channel, so that it will be called by the module when the channel detects an edge.

```
extint_chan_enable_callback(CONF_FAULT_EIC_LINE,
        EXTINT_CALLBACK_TYPE_DETECT);
```

Configure EVENTS for fault input

1. Create a event resource instance struct for the EVENTS module to store.

```
struct events_resource event_resource;
```

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

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

```
struct events_config config;
```

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

```
events_get_config_defaults(&config);
```

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

4. Adjust the configuration struct to request that the channel be attached to the specified event generator, and that the asynchronous event path be used. Here the EIC channel connected to board button is the event generator.

```
config.generator = CONF_FAULT_EVENT_GENERATOR;
config.path      = EVENTS_PATH_ASYNCHRONOUS;
```

5. Allocate and configure the channel using the configuration structure.

```
events_allocate(&event_resource, &config);
```

Note The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

6. Attach an user to the channel. Here the user is TCC channel 0 event, which has been configured as input of Recoverable Fault.

```
events_attach_user(&event_resource, CONF_FAULT_EVENT_USER);
```

8.6.2 Use Case

8.6.2.1 Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

while (true) {
}
```

8.6.2.2 Workflow

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

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

8.7 Quick Start Guide for Using DMA with TCC

The supported board list:

- SAM D21/R21 Xplained Pro

In this use case, the TCC will be used to generate a PWM signal. Here the pulse width varies through following values with the help of DMA transfer: one quarter of the period, half of the period, and three quarters of the period. The PWM output can be used to drive an LED. The waveform can also be viewed using an oscilloscope. The output signal is also fed back to another TCC channel by event system, the event stamps are captured and transferred to a buffer by DMA.

The PWM output is set up as follows:

Board	Pin	Connect to
SAMD21 Xpro	PB30	LED0
SAMR21 Xpro	PA19	LED0

The TCC module will be setup as follows:

- GCLK generator 0 (GCLK main) clock source
- Use double buffering write when set top, compare, or pattern through API
- No dithering on the counter or compare
- No prescaler
- Single Slope PWM wave generation
- GCLK reload action
- Don't run in standby

- No fault or waveform extensions
- No inversion of waveform output
- No capture enabled
- Count upward
- Don't perform one-shot operations
- Counter starts on 0
- Counter top set to 0x1000
- Channel 0 (on SAM D21 Xpro) or 3 (on SAM R21 Xpro) is set to compare and match value 0x1000*3/4 and generate event
- Channel 1 is set to capture on input event

The event resource of EVSYS module will be setup as follows:

- TCC match capture channel 0 (on SAM D21 Xpro) or 3 (on SAM R21 Xpro) is selected as event generator
- Event generation is synchronous, with rising edge detected
- TCC match capture channel 1 is the event user

The DMA resource of DMAC module will be setup as follows:

- Two DMA resources are used
- Both DMA resources use peripheral trigger
- Both DMA resources perform beat transfer on trigger
- Both DMA resources use beat size of 16 bits
- Both DMA resources are configured to transfer three beats and then repeat again in same buffer
- On DMA resource which controls the compare value
  - TCC0 overflow triggers DMA transfer
  - The source address increment is enabled
  - The destination address is fixed to TCC channel 0 Compare/Capture register
- On DMA resource which reads the captured value
  - TCC0 capture on channel 1 triggers DMA transfer
  - The source address is fixed to TCC channel 1 Compare/Capture register
  - The destination address increment is enabled
  - The captured value is transferred to an array in SRAM

8.7.1 Quick Start

8.7.1.1 Prerequisites

There are no prerequisites for this use case.

8.7.1.2 Code

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

- SAM D21 Xplained Pro.

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL      1
#define CONF_TCC_EVENT_GENERATOR      EVSYS_ID_GEN_TCC0_MCX_0
#define CONF_TCC_EVENT_USER           EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

```
#define CONF_CAPTURE_TRIGGER TCC0_DMAC_ID_MC_1
```

- SAM R21 Xplained Pro.

```
#define CONF_PWM_MODULE      LED_0_PWM4CTRL_MODULE
#define CONF_PWM_CHANNEL     LED_0_PWM4CTRL_CHANNEL
#define CONF_PWM_OUTPUT      LED_0_PWM4CTRL_OUTPUT
#define CONF_PWM_OUT_PIN     LED_0_PWM4CTRL_PIN
#define CONF_PWM_OUT_MUX     LED_0_PWM4CTRL_MUX
```

```
#define CONF_TCC_CAPTURE_CHANNEL      1
#define CONF_TCC_EVENT_GENERATOR      EVSYS_ID_GEN_TCC0_MCX_3
#define CONF_TCC_EVENT_USER           EVSYS_ID_USER_TCC0_MC_1
```

```
#define CONF_COMPARE_TRIGGER TCC0_DMAC_ID_OVF
```

```
#define CONF_CAPTURE_TRIGGER TCC0_DMAC_ID_MC_1
```

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

```
struct tcc_module tcc_instance;
```

```
uint16_t capture_values[3] = {0, 0, 0};
struct dma_resource capture_dma_resource;
COMPILER_ALIGNED(16) DmacDescriptor capture_dma_descriptor;
struct events_resource capture_event_resource;
```

```
uint16_t compare_values[3] = {
    (0x1000 / 4), (0x1000 * 2 / 4), (0x1000 * 3 / 4)
};
struct dma_resource compare_dma_resource;
COMPILER_ALIGNED(16) DmacDescriptor compare_dma_descriptor;
```

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

```
static void config_event_for_capture(void)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator      = CONF_TCC_EVENT_GENERATOR;
    config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
    config.path           = EVENTS_PATH_SYNCHRONOUS;
    config.clock_source   = GCLK_GENERATOR_0;

    events_allocate(&capture_event_resource, &config);

    events_attach_user(&capture_event_resource, CONF_TCC_EVENT_USER);
}
```

```
static void config_dma_for_capture(void)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
    config.peripheral_trigger = CONF_CAPTURE_TRIGGER;

    dma_allocate(&capture_dma_resource, &config);

    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = 3;
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.step_selection = DMA_STEPSEL_SRC;
    descriptor_config.src_increment_enable = false;
    descriptor_config.source_address =
            (uint32_t)&CONF_PWM_MODULE->CC[CONF_TCC_CAPTURE_CHANNEL];
    descriptor_config.destination_address =
            (uint32_t)capture_values + sizeof(capture_values);

    dma_descriptor_create(&capture_dma_descriptor, &descriptor_config);

    dma_add_descriptor(&capture_dma_resource, &capture_dma_descriptor);
    dma_add_descriptor(&capture_dma_resource, &capture_dma_descriptor);
    dma_start_transfer_job(&capture_dma_resource);
}
```

```
static void config_dma_for_wave(void)
```

```
{
    struct dma_resource_config config;
    dma_get_config_defaults(&config);
    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
    config.peripheral_trigger = CONF_COMPARE_TRIGGER;
    dma_allocate(&compare_dma_resource, &config);

    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.block_transfer_count = 3;
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.source_address =
            (uint32_t)compare_values + sizeof(compare_values);
    descriptor_config.destination_address =
            (uint32_t)&CONF_PWM_MODULE->CC[CONF_PWM_CHANNEL];

    dma_descriptor_create(&compare_dma_descriptor, &descriptor_config);

    dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
    dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
    dma_start_transfer_job(&compare_dma_resource);
}
```

```
static void configure_tcc(void)
{
    struct tcc_config config_tcc;
    tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);

    config_tcc.counter.period = 0x1000;
    config_tcc.compare.channel_function[CONF_TCC_CAPTURE_CHANNEL] =
            TCC_CHANNEL_FUNCTION_CAPTURE;
    config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
    config_tcc.compare.wave_polarity[CONF_PWM_CHANNEL] = TCC_WAVE_POLARITY_0;
    config_tcc.compare.match[CONF_PWM_CHANNEL] = compare_values[2];

    config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
    config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
    config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;

    tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);

    struct tcc_events events_tcc = {
        .input_config[0].modify_action = false,
        .input_config[1].modify_action = false,
        .output_config.modify_generation_selection = false,
        .generate_event_on_channel[CONF_PWM_CHANNEL] = true,
        .on_event_perform_channel_action[CONF_TCC_CAPTURE_CHANNEL] = true
    };
    tcc_enable_events(&tcc_instance, &events_tcc);

    config_event_for_capture();

    config_dma_for_capture();
    config_dma_for_wave();

    tcc_enable(&tcc_instance);
```

```
}
```

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

```
configure_tcc();
```

8.7.1.3 Workflow

Configure the TCC

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

```
struct tcc_module tcc_instance;
```

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

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

```
struct tcc_config config_tcc;
```

3. Initialize the TCC configuration struct with the module's default values.

```
tcc_get_config_defaults(&config_tcc, CONF_PWM_MODULE);
```

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

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

```
config_tcc.counter.period = 0x1000;
config_tcc.compare.channel_function[CONF_TCC_CAPTURE_CHANNEL] =
        TCC_CHANNEL_FUNCTION_CAPTURE;
config_tcc.compare.wave_generation = TCC_WAVE_GENERATION_SINGLE_SLOPE_PWM;
config_tcc.compare.wave_polarity[CONF_PWM_CHANNEL] = TCC_WAVE_POLARITY_0;
config_tcc.compare.match[CONF_PWM_CHANNEL] = compare_values[2];
```

5. Alter the TCC settings to configure the PWM output on a physical device pin.

```
config_tcc.pins.enable_wave_out_pin[CONF_PWM_OUTPUT] = true;
config_tcc.pins.wave_out_pin[CONF_PWM_OUTPUT]        = CONF_PWM_OUT_PIN;
config_tcc.pins.wave_out_pin_mux[CONF_PWM_OUTPUT]    = CONF_PWM_OUT_MUX;
```

6. Configure the TCC module with the desired settings.

```
tcc_init(&tcc_instance, CONF_PWM_MODULE, &config_tcc);
```

7. Configure and enable the desired events for the TCC module.

```
struct tcc_events events_tcc = {
    .input_config[0].modify_action = false,
    .input_config[1].modify_action = false,
    .output_config.modify_generation_selection = false,
    .generate_event_on_channel[CONF_PWM_CHANNEL] = true,
    .on_event_perform_channel_action[CONF_TCC_CAPTURE_CHANNEL] = true
};
tcc_enable_events(&tcc_instance, &events_tcc);
```

Configure the Event System

Configure the EVSYS module to wire channel 0 event to channel 1.

1. Create an event resource instance.

```
struct events_resource capture_event_resource;
```

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

2. Create an event resource configuration struct.

```
struct events_config config;
```

3. Initialize the event resource configuration struct with default values.

```
events_get_config_defaults(&config);
```

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

4. Adjust the event resource configuration to desired values.

```
config.generator      = CONF_TCC_EVENT_GENERATOR;
config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
config.path           = EVENTS_PATH_SYNCHRONOUS;
config.clock_source   = GCLK_GENERATOR_0;
```

5. Allocate and configure the resource using the configuration structure.

```
events_allocate(&capture_event_resource, &config);
```

6. Attach a user to the resource.

```
events_attach_user(&capture_event_resource, CONF_TCC_EVENT_USER);
```

Configure the DMA for Capture TCC Channel 1

Configure the DMAC module to obtain captured value from TCC channel 1.

1. Create a DMA resource instance.

```
struct dma_resource capture_dma_resource;
```

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

2. Create a DMA resource configuration struct.

```
struct dma_resource_config config;
```

3. Initialize the DMA resource configuration struct with default values.

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

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

4. Adjust the DMA resource configurations.

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

5. Allocate a DMA resource with the configurations.

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

6. Prepare DMA transfer descriptor.

   a. Create a DMA transfer descriptor.

```
COMPILER_ALIGNED(16) DmacDescriptor capture_dma_descriptor;
```

Note When multiple descriptors are linked, the linked item should never go out of scope before it is loaded (to DMA Write-Back memory section). In most cases, if more than one descriptors are used, they should be global except the very first one.

   b. Create a DMA transfer descriptor struct.

   c. 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;
```

   d. Initialize the DMA transfer descriptor configuration struct with default values.

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

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

e. Adjust the DMA transfer descriptor configurations.

```
descriptor_config.block_transfer_count = 3;
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.step_selection = DMA_STEPSEL_SRC;
descriptor_config.src_increment_enable = false;
descriptor_config.source_address =
        (uint32_t)&CONF_PWM_MODULE->CC[CONF_TCC_CAPTURE_CHANNEL];
descriptor_config.destination_address =
        (uint32_t)capture_values + sizeof(capture_values);
```

f. Create the DMA transfer descriptor with the given configuration.

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

7. Start DMA transfer job with prepared descriptor.

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

```
dma_add_descriptor(&capture_dma_resource, &capture_dma_descriptor);
dma_add_descriptor(&capture_dma_resource, &capture_dma_descriptor);
```

Note When adding multiple descriptors, the last added one is linked at the end of descriptor queue. If ringed list is needed, just add the first descriptor again to build the circle.

b. Start the DMA transfer job with the allocated DMA resource and transfer descriptor.

```
dma_start_transfer_job(&capture_dma_resource);
```

Configure the DMA for Compare TCC Channel 0

Configure the DMAC module to update TCC channel 0 compare value. The flow is similar to last DMA configure step for capture.

1. Allocate and configure the DMA resource.

```
struct dma_resource compare_dma_resource;
```

```
struct dma_resource_config config;
dma_get_config_defaults(&config);
config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
config.peripheral_trigger = CONF_COMPARE_TRIGGER;
dma_allocate(&compare_dma_resource, &config);
```

2. Prepare DMA transfer descriptor.

```
COMPILER_ALIGNED(16) DmacDescriptor compare_dma_descriptor;
```

```
struct dma_descriptor_config descriptor_config;

dma_descriptor_get_config_defaults(&descriptor_config);

descriptor_config.block_transfer_count = 3;
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.source_address =
        (uint32_t)compare_values + sizeof(compare_values);
descriptor_config.destination_address =
        (uint32_t)&CONF_PWM_MODULE->CC[CONF_PWM_CHANNEL];

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

3. Start DMA transfer job with prepared descriptor.

```
dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
dma_add_descriptor(&compare_dma_resource, &compare_dma_descriptor);
dma_start_transfer_job(&compare_dma_resource);
```

4. Enable the TCC module to start the timer and begin PWM signal generation.

```
tcc_enable(&tcc_instance);
```

8.7.2 Use Case

8.7.2.1 Code

Copy-paste the following code to your user application:

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

8.7.2.2 Workflow

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

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

Index

U

Union Definitions

Document Revision History

Doc. Rev.	Date	Comments
B	12/2014	Added fault handling functionality; Added double buffering functionality with use case; Added timer use case; Added SAM R21/ D10/D11 support
A	01/2014	Initial release

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