# EE 3171 Lecture 11

Tiva C Timers

# Real-Time

Real-Time (RT) -- The perfect time base of the cosmos (e.g., The Watchmaker's watch)

Ignoring relativistic and quantum effects, RT:

- Proceeds at a constant rate
- Is a continuous (non-discrete) function

RT can not be measured or read precisely

RT can only be approximated by a clock

- Clocks have finite precision (tick length)
- Clock accuracy (tick rate) varies over time
- Clock accuracies differ from each other

# Realistic Real-Time

A system that can guarantee an upper bound (worst case) on latency (response time) for time-critical operations.

Some people will talk of so-called soft real-time, in which there is only a “very good probability” operations will be completed within that upper bound on latency.

Most people who work with “hard” real-time systems do not consider “soft” real-time to be a RTS at all.

# Basic Timer Usage

Timers are used in microcontrollers to:

- Timestamp inputs
- Schedule outputs
- Create precise delays
- Measure duration of processes
- Count external events

The Tiva C has timers for all of these purposes

# TM4C123 Timers

The General-Purpose Timer Module (GPTM) contains:

Six 16/32-bit GPTM blocks and…

Six 32/64-bit Wide GPTM blocks

Each 16/32-bit GPTM block provides two 16-bit timers/counters (referred to as *Timer A* and *Timer B*) that can be configured to operate independently as:

Timers

Event counters

Concatenated to operate as one 32-bit timer

Concatenated to operate as one 32-bit Real-Time Clock (RTC)

Each 32/64-bit Wide GPTM block provides 32-bit timers for Timer A and Timer B that can be concatenated to operate as a 64-bit timer.

Timers can also be used to trigger µDMA transfers and ADC conversions

# 16/32 bit GPTM Functions

16- or 32-bit programmable one-shot timer **Count down once.**

16- or 32-bit programmable periodic timer **Count down lots.**

16-bit general-purpose timer with an 8-bit prescaler

32-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input

16-bit input-edge count or time-capture modes with an 8-bit prescaler

16-bit PWM mode with an 8-bit prescaler and software-programmable output inversion of the PWM signal

# 32/64 bit GPTM Functions

32- or 64-bit programmable one-shot timer

32- or 64-bit programmable periodic timer

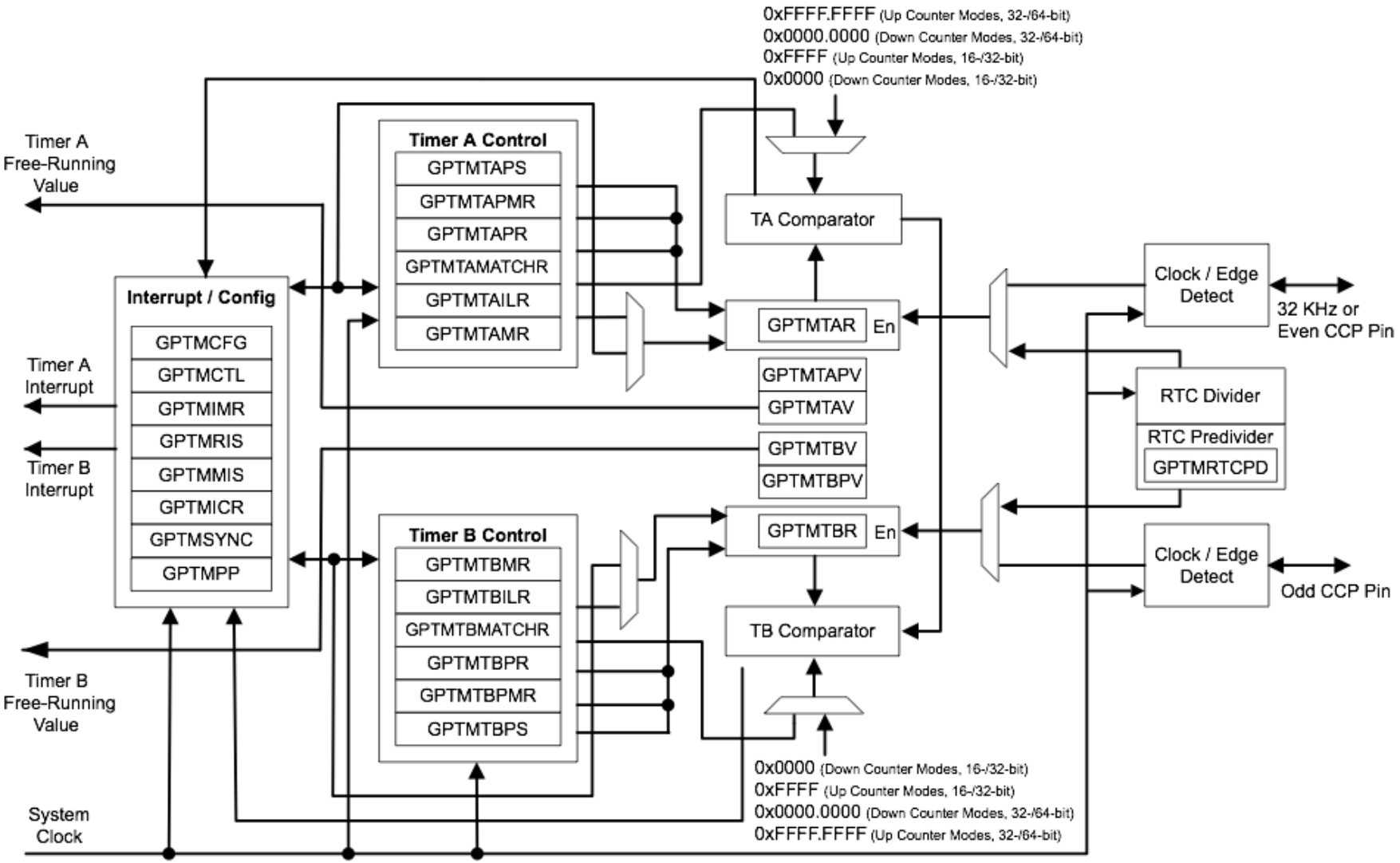
32-bit general-purpose timer with a 16-bit prescaler

64-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input

32-bit input-edge count- or time-capture modes with a16-bit prescaler

32-bit PWM mode with a 16-bit prescaler and software-programmable output inversion of the PWM signal

# Block Diagram



# A Different Approach

We've always examined a peripheral with a specific configuration in mind.

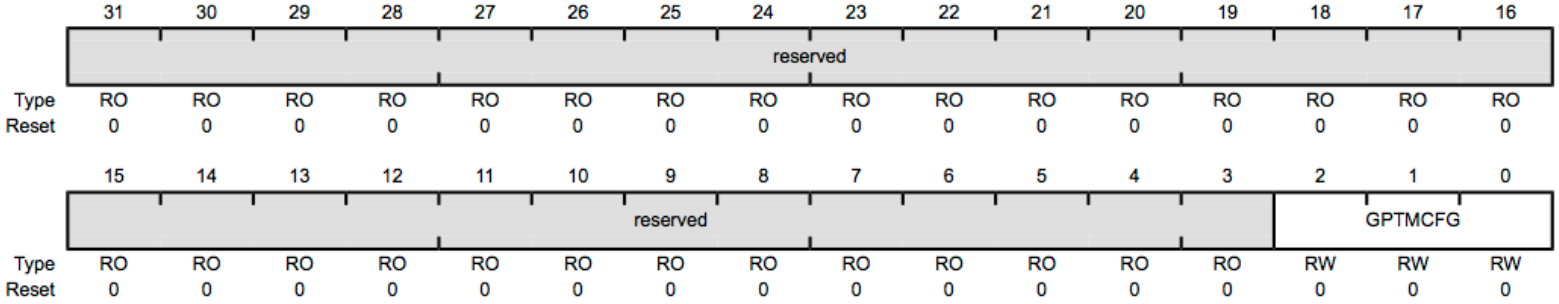
There are too many possible configurations for the timers, so we're going to look at the key registers, what control bits they contain and how to configure them for each mode.

As we go, we'll look at the TivaWare functions that accomplish the same thing.

# GPTM Configuration

**GPTM Configuration (GPTMCFG)**

16/32-bit Timer 0 base: 0x4003.0000
16/32-bit Timer 1 base: 0x4003.1000
16/32-bit Timer 2 base: 0x4003.2000
16/32-bit Timer 3 base: 0x4003.3000
16/32-bit Timer 4 base: 0x4003.4000
16/32-bit Timer 5 base: 0x4003.5000
32/64-bit Wide Timer 0 base: 0x4003.6000
32/64-bit Wide Timer 1 base: 0x4003.7000
32/64-bit Wide Timer 2 base: 0x4004.C000
32/64-bit Wide Timer 3 base: 0x4004.D000
32/64-bit Wide Timer 4 base: 0x4004.E000
32/64-bit Wide Timer 5 base: 0x4004.F000
Offset 0x000
Type RW, reset 0x0000.0000




Values for **GPTMCFG**:

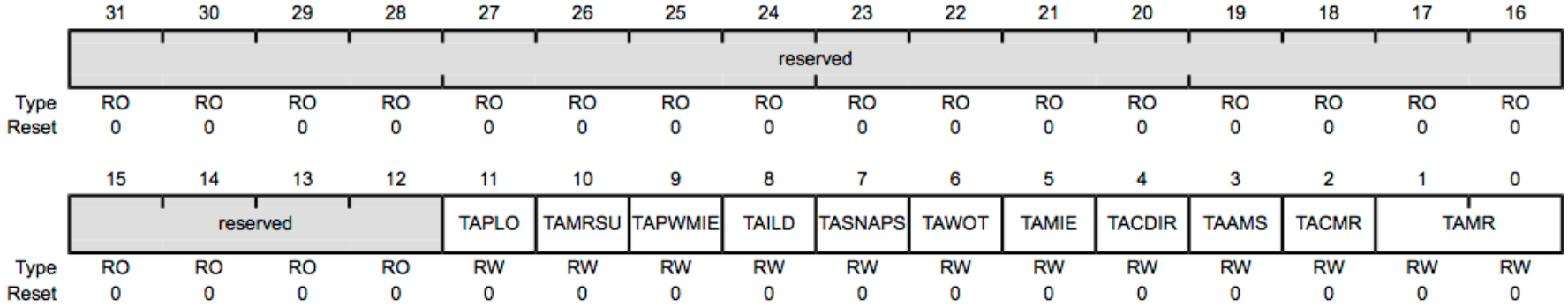
0x0 — Selects the “full” timer (32 or 64 bits)

0x1 — Selects “full” Real Time Clock

0x4 — Selects “half” timer (16 or 32 bits)

# GPTM Timer A Mode

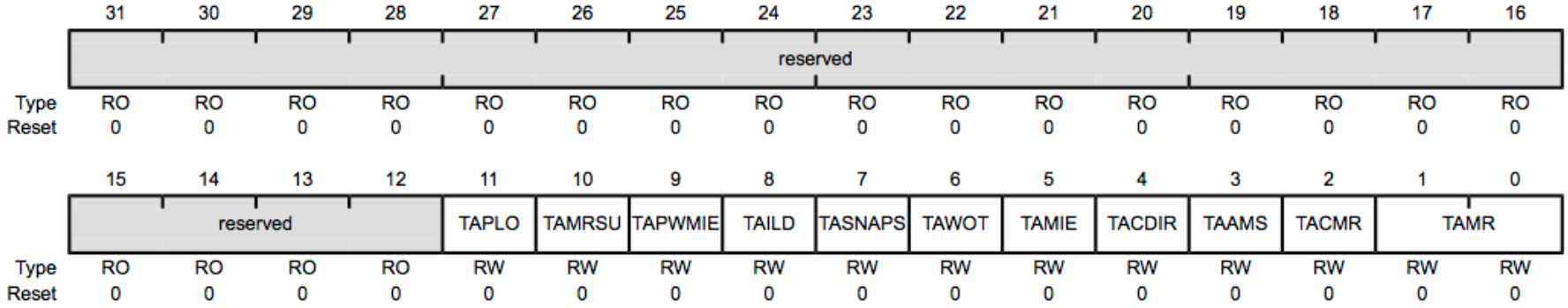
GPTM Timer A Mode (GPTMTAMR)




| Bit | 0 | 1 |
|---|---|---|
| TAMRSU | Update match register on next cycle | Update match registers on next timeout |
| TAPWMIE | Disable PWM Interrupts | Enable PWM Interrupts |
| TAILD | Update the counter registers on the next cycle | Update the counter registers on the next timeout |
| TASNAPS | Disable snapshot mode | Load Timer A register at snapshot event |
| TAWOT | Timer counts as soon as enabled | Timer waits for trigger event |

# GPTM Timer A Mode

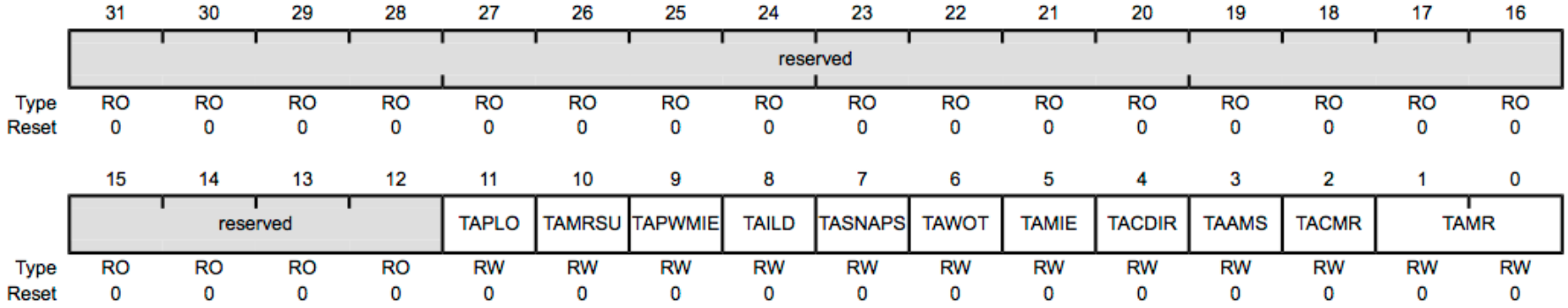
GPTM Timer A Mode (GPTMTAMR)




| Bit | 0 | 1 |
|---|---|---|
| TAMIE | Disable match interrupts | Enable match interrupts |
| TACDIR | Counts down | Counts up |
| TAAMS | Capture or compare mode | PWM mode |
| TACMR | Edge count capture mode | Edge time capture mode |

# GPTM Timer A Mode

GPTM Timer A Mode (GPTMTAMR)




| Bit | 0x0 | 0x1 | 0x2 | 0x3 |
|---|---|---|---|---|
| **TAMR** | Reserved | One-Shot Timer | Periodic Timer | Capture Mode |

# TivaWare Access to `GPTMTAMR`

Two functions:

- `TimerConfigure()` handles almost all of the access to `GPTMTAMR`.
- `TimerControlWaitOnTrigger()` handles exactly two bits, the Wait On Trigger bits.

# TimerConfigure()

Configures the timer(s).

```
void TimerConfigure(uint32_t ui32Base, uint32_t
ui32Config)
```

Parameters:

`ui32Base` is the base address of the timer module.
`ui32Config` is the configuration for the timer.

This function configures the operating mode of the timer(s). The timer module is disabled before being configured and is left in the disabled state. The timer can be configured to be a single full-width timer by using the `TIMER_CFG_*` values or a pair of half-width timers using the `TIMER_CFG_A_*` and `TIMER_CFG_B_*` values passed in the `ui32Config` parameter.

`ui32Config` parameter options on slide number next.

# TimerConfigure()

The configuration is specified in `ui32Config` as one of the following values:

`TIMER_CFG_ONE_SHOT` - Full-width one-shot timer
`TIMER_CFG_ONE_SHOT_UP` - Full-width one-shot timer that counts up instead of down (not available on all parts)
`TIMER_CFG_PERIODIC` - Full-width periodic timer
`TIMER_CFG_PERIODIC_UP` - Full-width periodic timer that counts up instead of down (not available on all parts)
`TIMER_CFG_RTC` - Full-width real time clock timer
`TIMER_CFG_SPLIT_PAIR` - Two half-width timers

When configured for a pair of half-width timers, each timer is separately configured. The first timer is configured by setting `ui32Config` to the result of a logical OR operation between one of the following values and `ui32Config`:

`TIMER_CFG_A_ONE_SHOT` - Half-width one-shot timer
`TIMER_CFG_A_ONE_SHOT_UP` - Half-width one-shot timer that counts up instead of down (not available on all parts)
`TIMER_CFG_A_PERIODIC` - Half-width periodic timer
`TIMER_CFG_A_PERIODIC_UP` - Half-width periodic timer that counts up instead of down (not available on all parts)
`TIMER_CFG_A_CAP_COUNT` - Half-width edge count capture
`TIMER_CFG_A_CAP_TIME` - Half-width edge time capture
`TIMER_CFG_A_CAP_TIME_UP` - Half-width edge time capture that counts up instead of down (not available on all parts)
`TIMER_CFG_A_PWM` - Half-width PWM output

There are `TIMER_CFG_B_*` values too.

# TimerControlWaitOnTrigger()

Controls the wait on trigger handling.

```
void TimerControlWaitOnTrigger(uint32_t ui32Base,
uint32_t ui32Timer, bool bWait)
```

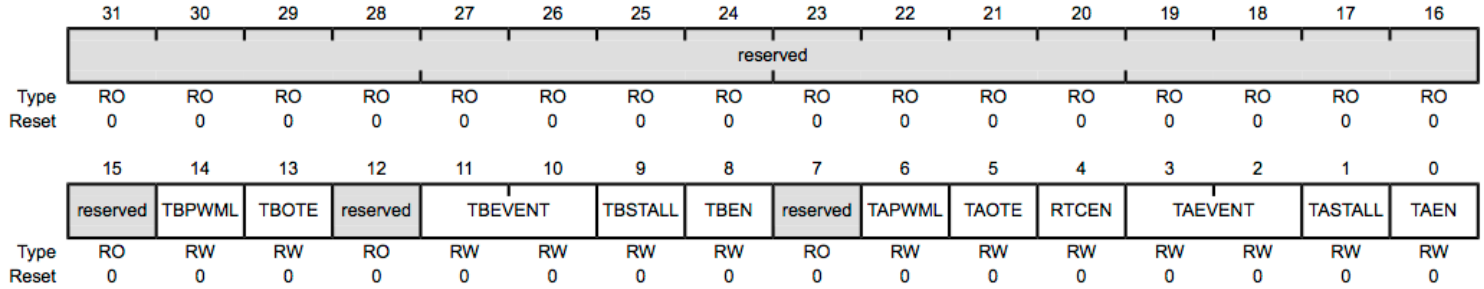
Parameters:

`ui32Base` is the base address of the timer module.
`ui32Timer` specifies the timer(s) to be adjusted;
must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.
`bWait` specifies if the timer should wait for a trigger input.

This function controls whether or not a timer waits for a trigger input to start counting. When enabled, the previous timer in the trigger chain must count to its timeout in order for this timer to start counting.

# GPTM Control

GPTM Control (GPTMCTL)




| Bit | 0 | 1 |
|---|---|---|
| TBPWML | PWM Output Normal | PWM Output Inverted |
| TBOTE | Timer B ADC Trigger Disabled | Timer B ADC Trigger Enabled |
| TBEVENT | 0x0: Positive Edge Triggered<br>0x01: Negative Edge Trigger | 0x3: Both Edges |
| TBSTALL | Timer continues even when debugger halts processor | Timer stops when debugger halts processor |
| TBEN | Timer B Disabled | Timer B Enabled |

The other bits are just for Timer A.

# TivaWare Access to GPTMCTL

Lots of functions:

`TimerControlEvent()` configures the edge that triggers a capture event.

`TimerControlLevel()` configures the polarity of the PWM signal.

`TimerControlStall()` configures the `T*STALL` bits.

`TimerControlTrigger()` configures the ADC output trigger.

`TimerDisable()` disables a timer.

`TimerEnable()` enables a timer.

# TimerControlEvent()

Controls the event type.

```
void TimerControlEvent(uint32_t ui32Base, uint32_t ui32Timer,
uint32_t ui32Event)
```

Parameters:

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer(s) to be adjusted;

Must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

`ui32Event` specifies the type of event;

Must be one of `TIMER_EVENT_POS_EDGE`, `TIMER_EVENT_NEG_EDGE`, or `TIMER_EVENT_BOTH_EDGES`.

This function configures the signal edge(s) that triggers the timer when in capture mode.

# TimerControlLevel()

Controls the output level.

```
void TimerControlLevel(uint32_t ui32Base, uint32_t
ui32Timer, bool bInvert)
```

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer(s) to adjust;

must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

`bInvert` specifies the output level.

This function configures the PWM output level for the specified timer. If the `bInvert` parameter is true, then the timer's output is made active low; otherwise, it is made active high.

# TimerControlStall()

Controls the stall handling.

```
void TimerControlStall(uint32_t ui32Base, uint32_t
ui32Timer, bool bStall)
```

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer(s) to be adjusted;

must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

`bStall` specifies the response to a stall signal.

This function controls the stall response for the specified timer. If the `bStall` parameter is true, then the timer stops counting if the processor enters debug mode; otherwise the timer keeps running while in debug mode.

# TimerControlTrigger()

Enables or disables the ADC trigger output.

```
void TimerControlTrigger(uint32_t ui32Base, uint32_t
ui32Timer, bool bEnable)
```

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer to adjust;

must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

`bEnable` specifies the desired ADC trigger state.

This function controls the ADC trigger output for the specified timer. If the `bEnable` parameter is true, then the timer's ADC output trigger is enabled; otherwise it is disabled.

# TimerDisable()

Disables the timer(s).

```
void TimerDisable(uint32_t ui32Base,
uint32_t ui32Timer)
```

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer(s) to disable;

must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

This function disables operation of the timer module.

# TimerEnable()

Enables the timer(s).

```
void TimerEnable(uint32_t ui32Base,
uint32_t ui32Timer)
```

`ui32Base` is the base address of the timer module.

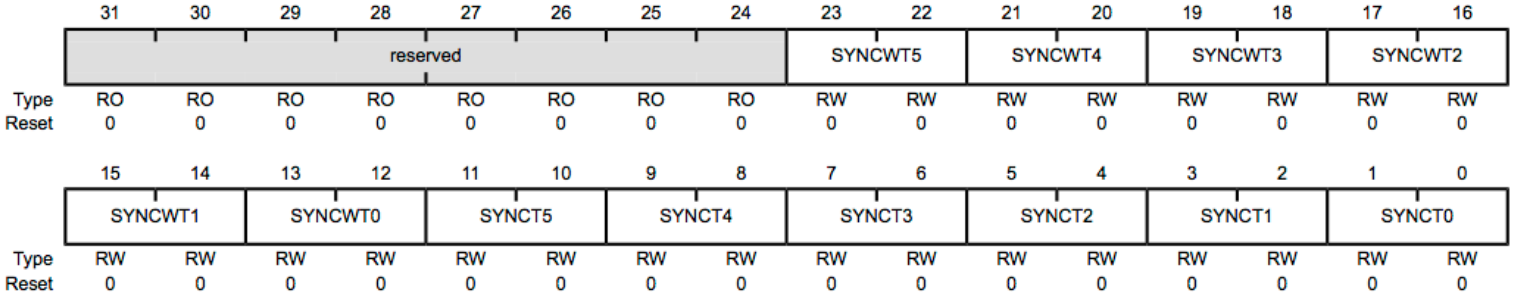
`ui32Timer` specifies the timer(s) to disable;

must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

This function enables operation of the timer module.

# GPTM Sync

GPTM Synchronize (GPTMSYNC)




| Bit | 0x0 | 0x1 | 0x2 | 0x3 |
|---|---|---|---|---|
| SYNCWT*x* | No effect | Timeout for Timer A of 32/64 bit timer *x* | Timeout for Timer B of 32/64 bit timer *x* | Timeout for both Timer A & B of 32/64 bit timer *x* |

Use these bits to make two counters start counting at the same time.

# TimerSynchronize()

Synchronizes the counters in a set of timers.

`void TimerSynchronize(uint32_t ui32Base, uint32_t ui32Timers)`

Parameters:

`ui32Base` is the base address of the timer module.

This parameter must be the base address of Timer0 (in other words, `TIMER0_BASE`).

`ui32Timers` is the set of timers to synchronize.

This function synchronizes the counters in a specified set of timers. When a timer is running in half-width mode, each half can be included or excluded in the synchronization event. When a timer is running in full-width mode, only the A timer can be synchronized (specifying the B timer has no effect).

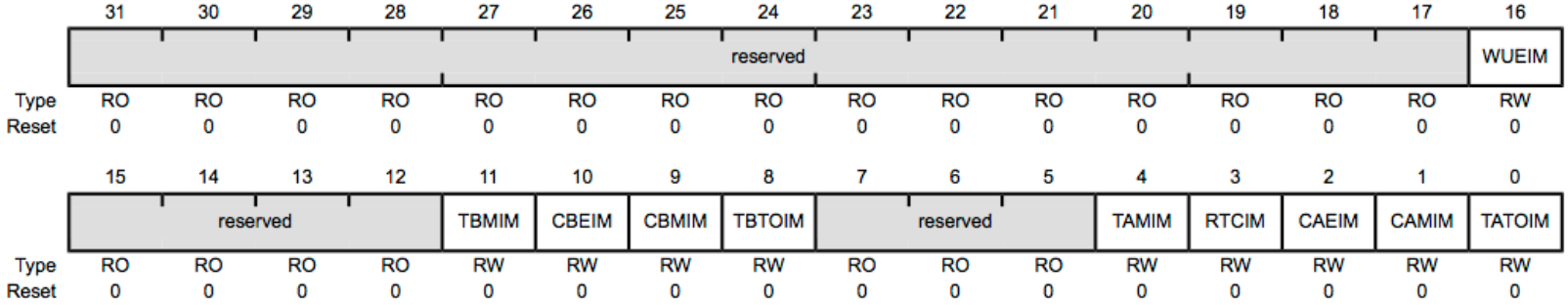
The `ui32Timers` parameter is the logical OR of defines of the form:

`TIMER_xA_SYNC`
`TIMER_xB_SYNC`
`WTIMER_xA_SYNC`
`WTIMER_xB_SYNC`

Obviously, replace *x* with a value between 0—5.

# GPTM Interrupt Mask

GPTM Interrupt Mask (GPTMIMR)




| Bit | 0 | 1 |
|---|---|---|
| WUEIM | Write Update Error Interrupt Disabled | Write Update Error Interrupt Enabled |
| TxMIM | Timer x Match Interrupt Disabled | Timer x Match Interrupt Enabled |
| CxEIM | Timer x Capture Event Interrupt Disabled | Timer x Capture Event Interrupt Enabled |
| CxMIM | Timer x Capture Match Interrupt Disabled | Timer x Capture Match Interrupt Enabled |
| TxTOIM | Timer x Time Out Interrupt Disabled | Timer x Time Out Interrupt Enabled |

*x* is A or B

# TimerIntEnable()

Enables individual timer interrupt sources.

```
void TimerIntEnable(uint32_t ui32Base, uint32_t ui32IntFlags)
```

Parameters:

`ui32Base` is the base address of the timer module.
`ui32IntFlags` is the bit mask of the interrupt sources to be enabled.

This function enables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The `ui32IntFlags` parameter must be the logical OR of any combination of the following:

`TIMER_CAPB_EVENT` - Capture B event interrupt
`TIMER_CAPB_MATCH` - Capture B match interrupt
`TIMER_TIMB_TIMEOUT` - Timer B timeout interrupt
`TIMER_RTC_MATCH` - RTC interrupt mask
`TIMER_CAPA_EVENT` - Capture A event interrupt
`TIMER_CAPA_MATCH` - Capture A match interrupt
`TIMER_TIMA_TIMEOUT` - Timer A timeout interrupt Returns:

# TimerIntDisable()

Disables individual timer interrupt sources.

`void TimerIntDisable(uint32_t ui32Base, uint32_t ui32IntFlags)`

Parameters:

`ui32Base` is the base address of the timer module.
`ui32IntFlags` is the bit mask of the interrupt sources to be disabled.

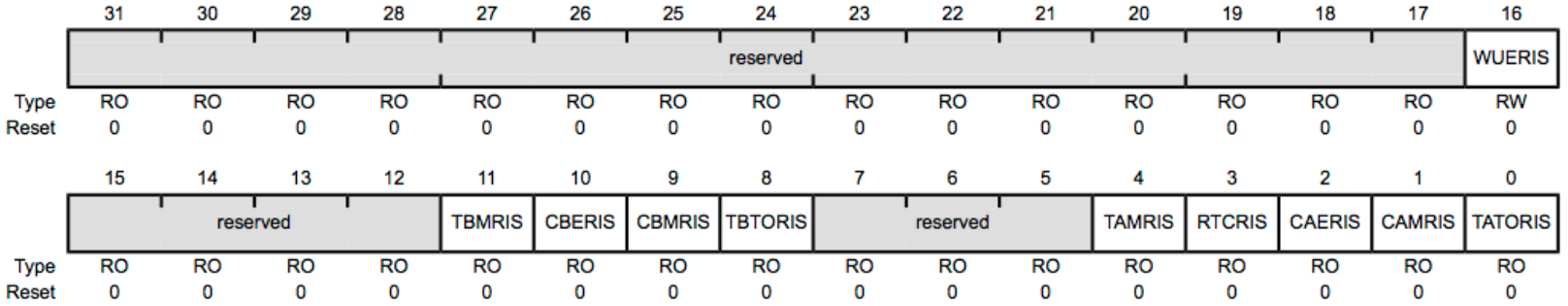
This function disables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The `ui32IntFlags` parameter must be the logical OR of any combination of the following:

`TIMER_CAPB_EVENT` - Capture B event interrupt
`TIMER_CAPB_MATCH` - Capture B match interrupt
`TIMER_TIMB_TIMEOUT` - Timer B timeout interrupt
`TIMER_RTC_MATCH` - RTC interrupt mask
`TIMER_CAPA_EVENT` - Capture A event interrupt
`TIMER_CAPA_MATCH` - Capture A match interrupt
`TIMER_TIMA_TIMEOUT` - Timer A timeout interrupt Returns:

# GPTM Interrupt Status

GPTM Raw Interrupt Status (GPTMRIS)

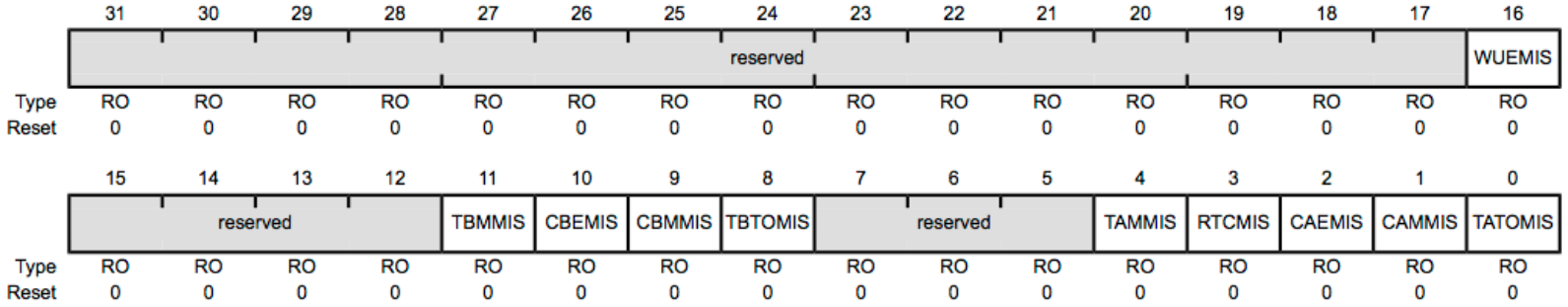



| Bit | 0 | 1 |
|---|---|---|
| WUERIS | No Event | Write Update Error Interrupt Fired |
| TxMRIS | No Event | Timer x Match Interrupt Fired |
| CxERIS | No Event | Timer x Capture Event Interrupt Fired |
| CxMRIS | No Event | Timer x Capture Match Interrupt Fired |
| TxTORIS | No Event | Timer x Time Out Interrupt Fired |

x is A or B

# GPTM Interrupt Status

GPTM Masked Interrupt Status (GPTMMIS)




| Bit | 0 | 1 |
|---|---|---|
| WUEMIS | No Event | Write Update Error Interrupt Fired |
| TxMMIS | No Event | Timer x Match Interrupt Fired |
| CxEMIS | No Event | Timer x Capture Event Interrupt Fired |
| CxMMIS | No Event | Timer x Capture Match Interrupt Fired |
| TxTOMIS | No Event | Timer x Time Out Interrupt Fired |

*x* is A or B

# TimerIntStatus()

Gets the current interrupt status.

`uint32_t TimerIntStatus(uint32_t ui32Base, bool bMasked)`

Parameters:

> `ui32Base` is the base address of the timer module.
> `bMasked` is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

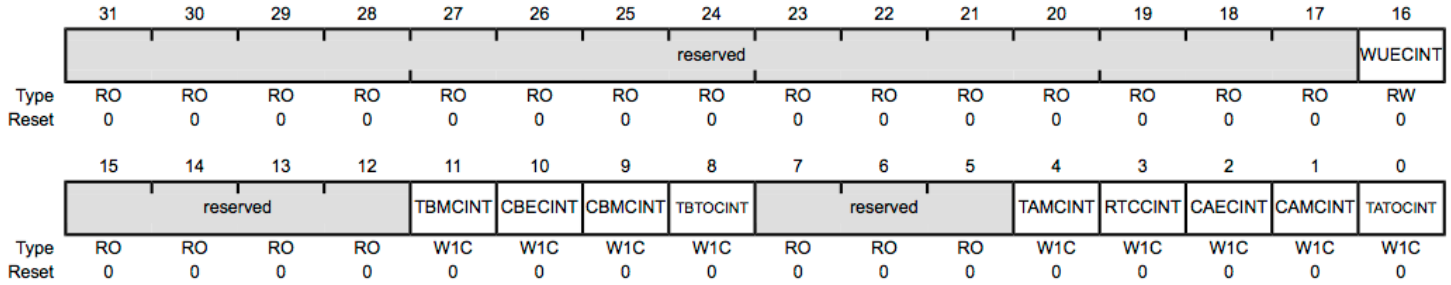
This function returns the interrupt status for the timer module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

The return value will be one of the following:

> `TIMER_CAPB_EVENT` - Capture B event interrupt
> `TIMER_CAPB_MATCH` - Capture B match interrupt
> `TIMER_TIMB_TIMEOUT` - Timer B timeout interrupt
> `TIMER_RTC_MATCH` - RTC interrupt mask
> `TIMER_CAPA_EVENT` - Capture A event interrupt
> `TIMER_CAPA_MATCH` - Capture A match interrupt
> `TIMER_TIMA_TIMEOUT` - Timer A timeout interrupt Returns:

# GPTM Interrupt Clear

GPTM Interrupt Clear (GPTMICR)




| Bit | 0 | 1 |
|---|---|---|
| WUEMIS | No Effect | Write 1 to clear Update Error Interrupt |
| TxMMIS | No Effect | Write 1 to clear Timer x Match Interrupt |
| CxEMIS | No Effect | Write 1 to clear Timer x Capture Event Interrupt |
| CxMMIS | No Effect | Write 1 to clear Timer x Capture Match Interrupt |
| TxTOMIS | No Effect | Write 1 to clear Timer x Time Out Interrupt |

x is A or B

# TimerIntClear()

Clears timer interrupt sources.

```
void TimerIntClear(uint32_t ui32Base, uint32_t ui32IntFlags)
```

Parameters:

`ui32Base` is the base address of the timer module.
`ui32IntFlags` is a bit mask of the interrupt sources to be cleared.

**TI recommends that you call this as early as possible in the ISR.**

Description:

The specified timer interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The `ui32IntFlags` value should be the logical OR of the following:

`TIMER_CAPB_EVENT` - Capture B event interrupt
`TIMER_CAPB_MATCH` - Capture B match interrupt
`TIMER_TIMB_TIMEOUT` - Timer B timeout interrupt
`TIMER_RTC_MATCH` - RTC interrupt mask
`TIMER_CAPA_EVENT` - Capture A event interrupt
`TIMER_CAPA_MATCH` - Capture A match interrupt
`TIMER_TIMA_TIMEOUT` - Timer A timeout interrupt Returns:

# Registers That Hold Counter Values

GPTM Timer A Interval Load (`GPTMTAILR`) — Contains the value loaded into the timer when counting down OR the upper bound when counting up.

GPTM Timer B Interval Load (`GPTMTBILR`) — Same as above, except Timer B.

GPTM Timer A Match (`GPTMTAMATCHR`) — When Timer A == Match, interrupt fires.

GPTM Timer B Match (`GPTMTBMATCHR`) — Same as above, except Timer B.

GPTM Timer A (`GPTMTAR`) — Current value of Timer A (or number of input edges)

GPTM Timer B (`GPTMTBR`) — Current value of Timer B (or number of input edges)

GPTM Timer A Value (`GPTMTAV`) — Current value of free-running Timer A in all modes.

GPTM Timer B Value (`GPTMTBV`) — Current value of free-running Timer B in all modes.

# Functions That Access Counter Registers

`TimerLoadSet()`

`TimerLoadGet()`

`TimerMatchSet()`

`TimerMatchGet()`

`TimerValueGet()`

Note no `TimerValueSet()` because that doesn't make any kind of sense.

# TimerLoadSet()

Sets the timer load value.

```
void TimerLoadSet(uint32_t ui32Base, uint32_t ui32Timer,
uint32_t ui32Value)
```

Parameters:

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer(s) to adjust; must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`. Only `TIMER_A` should be used when the timer is configured for full-width operation.

`ui32Value` is the load value.

This function configures the timer load value; if the timer is running then the value is immediately loaded into the timer.

# TimerLoadGet()

Gets the timer load value.

```
uint32_t TimerLoadGet(uint32_t ui32Base, uint32_t
ui32Timer)
```

Parameters:

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer; must be one of `TIMER_A` or `TIMER_B`. Only `TIMER_A` should be used when the timer is configured for full-width operation.

This function gets the currently programmed interval load value for the specified timer.

# TimerMatchSet()

Sets the timer match value.

```
void TimerMatchSet(uint32_t ui32Base, uint32_t ui32Timer,
uint32_t ui32Value)
```

Parameters:

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer(s) to adjust; must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`. Only `TIMER_A` should be used when the timer is configured for full-width operation.

`ui32Value` is the match value.

This function configures the match value for a timer. This value is used in capture count mode to determine when to interrupt the processor and in PWM mode to determine the duty cycle of the output signal. On some Tiva devices, match interrupts can also be generated in periodic and one-shot modes.

# TimerMatchGet()

Gets the timer match value.

```
uint32_t TimerMatchGet(uint32_t ui32Base,
uint32_t ui32Timer)
```

Parameters:

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer; must be one of `TIMER_A` or `TIMER_B`. Only `TIMER_A` should be used when the timer is configured for full-width operation.

This function gets the match value for the specified timer.

# TimerValueGet()

Gets the current timer value.

```
uint32_t TimerValueGet(uint32_t ui32Base,
uint32_t ui32Timer)
```

Parameters:

`ui32Base` is the base address of the timer module.

`ui32Timer` specifies the timer; must be one of `TIMER_A` or `TIMER_B`. Only `TIMER_A` should be used when the timer is configured for full-width operation.

This function reads the current value of the specified timer.

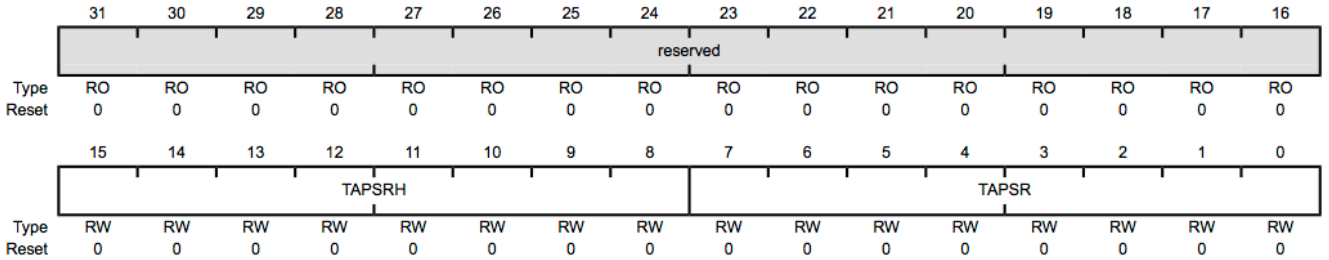
# Too Fast?

What if the counter is going too fast for your application?  How can you slow it down?

That's what the prescale registers are for.

# GPTM Timer Prescale

GPTM Timer A Prescale (GPTMTAPR)




| Bit | Values |
|---|---|
| TAPSRH | High-order byte for 16-bit prescaler |
| TAPSR | Prescaler for 8-bit prescale |

**What happens is that TAPSR counts down to 0 before the timer increments by 1. Later, rinse, repeat.**

# TimerPrescaleSet()

Set the timer prescale value.

```
void TimerPrescaleSet(uint32_t ui32Base, uint32_t ui32Timer,
uint32_t ui32Value)
```

Parameters:

`ui32Base` is the base address of the timer module.

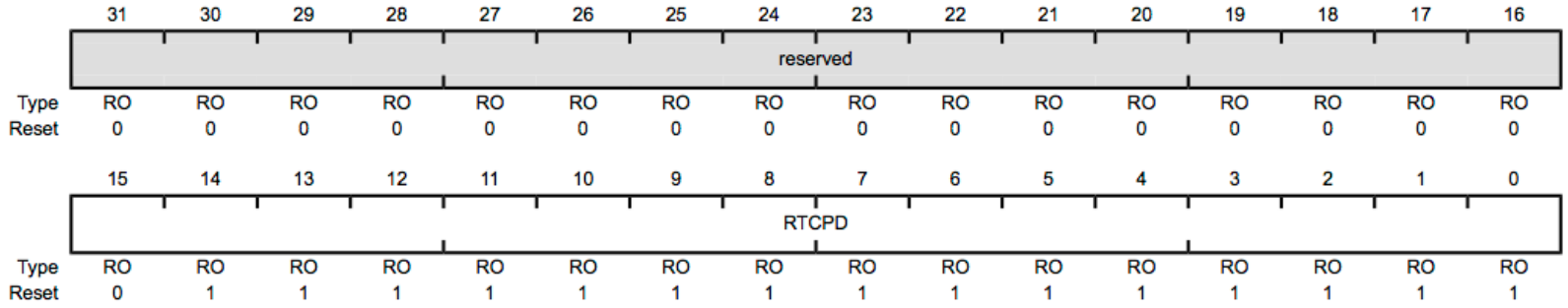
`ui32Timer` specifies the timer(s) to adjust; must be one of `TIMER_A`, `TIMER_B`, or `TIMER_BOTH`.

`ui32Value` is the timer prescale value which must be between 0 and 255 (inclusive) for 16/32-bit timers and between 0 and 65535 (inclusive) for 32/64-bit timers.

This function configures the value of the input clock prescaler. The prescaler is only operational when in half-width mode and is used to extend the range of the half-width timer modes. The prescaler provides the least significant bits when counting down in periodic and one-shot modes; in all other modes, the prescaler provides the most significant bits.

# GPTM RTC Prescale

GPTM RTC Predivide (GPTMRTCPD)




| Bit | Values |
|---|---|
| RTCPD | Predivider Value for RTC Clock |

# TimerRTCEnable()

Enable RTC counting.

```
void TimerRTCEnable(uint32_t ui32Base)
```

Parameters:

`ui32Base` is the base address of the timer module.

This function causes the timer to start counting when in RTC mode. If not configured for RTC mode, this function does nothing.

# TimerRTCDisable()

Disable RTC counting.

```
void TimerRTCDisable(uint32_t ui32Base)
```

Parameters:

`ui32Base` is the base address of the timer module.

This function causes the timer to stop counting when in RTC mode. If not configured for RTC mode, this function does nothing.

# Application 1: Counting Cars

We'll be taking an incoming signal and count the pulses.

# A Little Code

Configuration Code:

```
TimerConfigure(TIMER0_BASE, (TIMER_CFG_SPLIT_PAIR |
TIMER_CFG_B_CAP_COUNT));

TimerControlEvent(TIMER0_BASE, TIMER_B,
TIMER_EVENT_POS_EDGE);

TimerIntEnable(TIMER0_BASE, TIMER_CAPB_EVENT);

TimerEnable(TIMER0_BASE, TIMER_B);
```

Get the car count (and clear interrupt):

```
uint32_t carCount = TimerValueGet(TIMER0_BASE, TIMER_B);

TimerIntClear(TIMER0_BASE, TIMER_CAPB_EVENT);
```

# Application 2:Blinky

Configure Timer A to be a periodic timer to turn an LED off and on.

# A Little Code

Configuration Code:

```
TimerConfigure(TIMER0_BASE, (TIMER_CFG_SPLIT_PAIR |
TIMER_CFG_A_PERIODIC));

TimerLoadSet(TIMER0_BASE, TIMER_A, 3000);

TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

TimerEnable(TIMER0_BASE, TIMER_A);
```

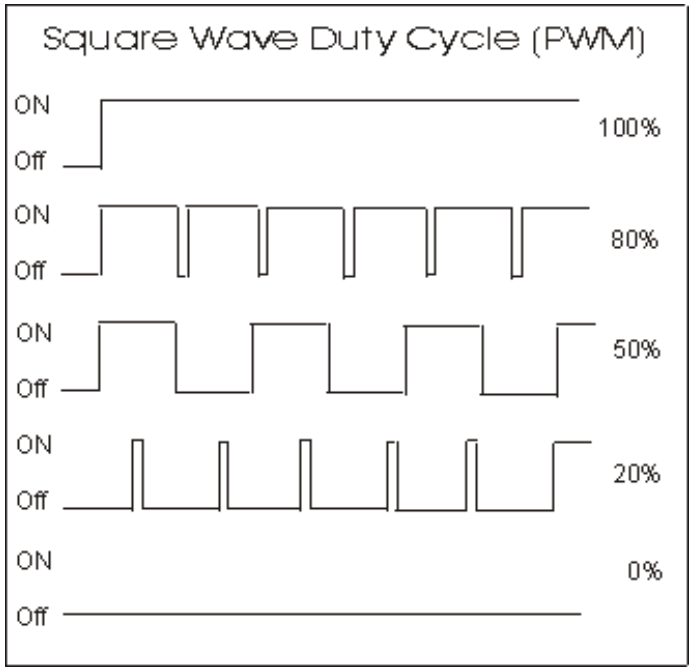
Toggle the LED in the ISR (and clear the interrupt):

```
LEDToggle(); // A complete cheater function

TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
```

# Application 3: PWM Detection

Use Timers A and B together to determine period and duty cycle of a signal.



# A Little Code

Configuration Code:

```
TimerConfigure(TIMER0_BASE, (TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_CAP_TIME |
TIMER_CFG_B_CAP_TIME));

TimerControlEvent(TIMER0_BASE, TIMER_A, TIMER_EVENT_BOTH_EDGES);

TimerControlEvent(TIMER0_BASE, TIMER_B, TIMER_EVENT_POS_EDGE);

TimerIntEnable(TIMER0_BASE, TIMER_TIMA_EVENT);

TimerIntEnable(TIMER0_BASE, TIMER_TIMB_EVENT);

TimerEnable(TIMER0_BASE, TIMER_A);

TimerEnable(TIMER0_BASE, TIMER_B);
```

Calculate DC in the ISR (and clear the interrupt):

```
uint32_t period = TimerValueGet(TIMER0_BASE, TIMER_B);
uint32_t highTime = TimerValueGet(TIMER0_BASE, TIMER_A);
float dc = highTime / period;
TimerIntClear(TIMER0_BASE, TIMER_CAPA_EVENT);
TimerIntClear(TIMER0_BASE, TIMER_CAPB_EVENT);
```

# Summary Slide

Microcontrollers use timers to:

Keep track of *when* something happened

*How long* it happened for

*How many* times it happened

Or to schedule events for the future