16-bit TachTimer Datasheet TachTimer16 V 1.1

Copyright © 2008-2014 Cypress Semiconductor Corporation. All Rights Reserved.

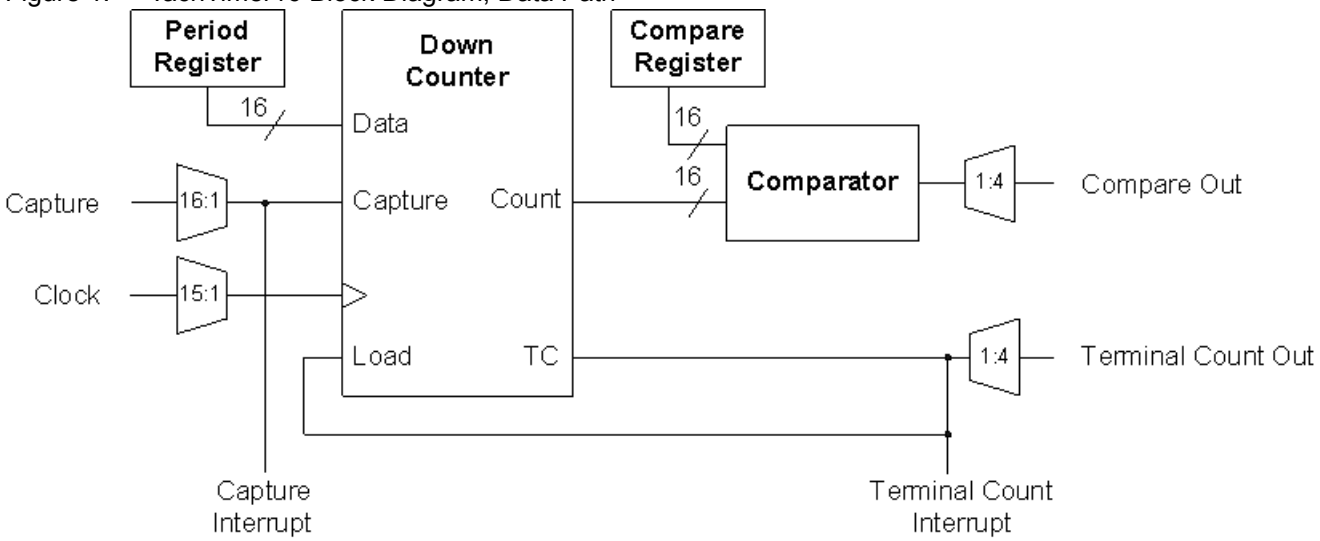
Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	flash	RAM	
CY8C29/27/24/21xxx, CY8CLED02/04/08/16, CY8CLED0xD, CY8CLED0xG, CY8C28x45, CY8CPLC20, CY8CLED16P01, CY8C28xxx	2	0	0	120	0	1

Features and Overview

- 16-bit timer uses two PSoC blocks
- Source clock rates up to 24 MHz
- Automatic reload of period on terminal count
- Use terminal count output pulse as input clock for other analog and digital functions
- Interrupt on terminal count and capture

The16-bit TachTimer User Module provides down counters with programmable period and capture ability. You can select the clock and enable signals from any system time base or external source. Once started, the timer operates continuously and reloads its internal value from the period register when it reaches terminal count. The output pulses high in the clock cycle after terminal count. Events can capture the current TachTimer16 count value by asserting the edge sensitive capture input signal. In each clock cycle, the TachTimer16 tests the count against the value of the compare register for either a less-than or less-than-or-equal-to condition. Interrupts are generated based upon terminal count and compare signals. The compare signal may be routed onto the row buses. The main between the TachTimer16 User Module and the Timer16 User Module is that the TachTimer16 provides **both** terminal count and capture interrupts. Timer16 gives you a choice of one or the other, but not both. The MSB block is the source for terminal count interrupt and the LSB block provides the capture interrupt. Another difference between the Timer 16 and the TachTimer16 is that the TachTimer16 has no parameter to choose the interrupt type. Interrupt types for both MSB and LSB are hard coded.

Figure 1. TachTimer16 Block Diagram, Data Path



Functional Description

The TachTimer16 User Module uses two digital PSoC blocks, each contributing 8 bits to the total resolution. To form timers that exceed 8 bits, consecutive blocks are linked so their internal carry, terminal count, and compare signals are synchronously chained. This concatenates the 8-bit Count, Period, and Compare registers (Data registers DR0, DR1 and DR2, respectively) from block-to-block to provide the 16-bit resolution. In this way, the TachTimer16 operates as a single monolithic synchronous device.

The TachTimer16 API provides functions callable from C and assembly languages to stop and start the operation of the TachTimer and to read and write the various data registers. A Control register starts and stops the TachTimer16 User Module. Writing the Period register, while the TachTimer16 is stopped copies the Count register into the Period register value. While the TachTimer16 is stopped, the output asserts low.

When a TachTimer16 is started, the Count register decrements by one on each rising edge of the clock. On the rising clock edge following the zero count, the Count register reloads from the Period register. On the next falling edge, the terminal count event is triggered and the output asserts high for one-half clock cycle.

You can modify the Period register with a new period value at any time. The Period register is automatically copied into the Count register when in the cycle after the value of the Count register reaches zero (terminal count).

The output period of the TachTimer16 User Module is effectively the period value programmed in the Period register, plus one.

Equation 1

$$OutputPeriod = PeriodValue + 1$$

The Period register value can be assigned in the Device Editor or at run time using the API.

On every input clock, the count in the Count register is compared to the value stored in the Compare register. The comparison performs a less-than or less-than-or-equal-to test, according to an option assigned to the CompareType parameter in the Device Editor. When the comparison condition is met, a compare event is triggered on the next clock. The TachTimer16 User Module provides the Compare output signal as an auxiliary output. This active high signal asserts on the rising edge of the clock cycle following

the cycle in which compare condition is satisfied. Specify this condition as **Less Than** or as **Less Than or Equal To**.

When the edge sensitive capture input is asserted high, the count value in the Count register transfers to the Compare register, causing a compare event on the next clock cycle. The Count value is then read using the ReadTimer API function. The elapsed time is computed as follows.

Equation 2

$$ElapsedTime = ClockPeriod \times (PeriodValue - CounterValue)$$

An interrupt is triggered on the terminal count and on capture events.The TachTimer16_LSB block provides the interrupt on capture event. The TachTimer16_MSB block provides the interrupt on the terminal count event.

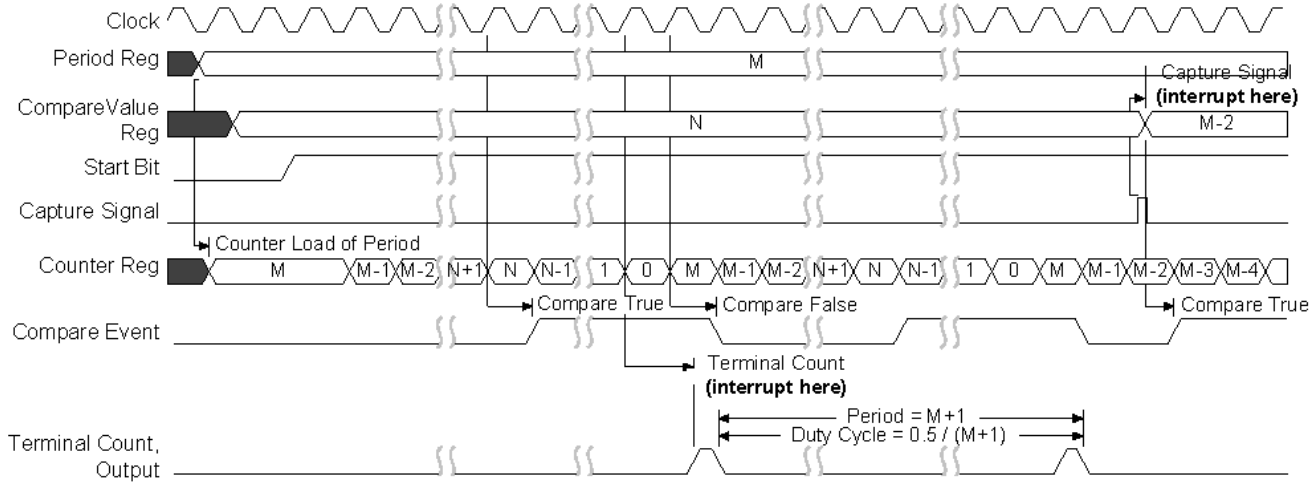
Using the ReadTimer() API function, a software algorithm triggers the capture mechanism to transfer the contents of the Count register into the Compare register. The Count register value is then returned.

For algorithms that require reading the timer countdown value on-the-fly without affecting the Count or Compare registers, call the ReadTimerSaveCV() API. This function reads the Count register value while preserving the Compare register contents. This function has some potential latency side effects.

Timing

External pins, routed to the counter by the global bus feature of the PSoC device, can clock the TachTimer16. The following figure illustrates the timing for the TachTimer16 User Modules.

Figure 2. TachTimer16 Timing Diagram



DC and AC Electrical Characteristics

Table 1. Timer AC Electrical Characteristics

Parameter	Typical	Limit	Units	Conditions and Notes
Maximum input frequency	--	24[ab]	MHz	Vdd=5.0V
Maximum output frequency	--	12[a]	MHz	Vdd=5.0V and 24 MHz input clock
	--	12[c]	MHz	Vdd=3.3V and 24 MHz input clock

a. If the input or output is routed through the global buses the frequency is limited to a maximum of 12 MHz.

b. For timer with an active capture function the input clock frequency limit is 24 MHz.

c. The fastest clock available to PSoC blocks is 24 MHz at 3.3V operation.

Placement

The TachTimer16 consumes two digital PSoC blocks that are placed consecutively by the Device Editor in order of increasing block number from least-significant byte (LSB) to most significant (the MSB). Each block is given a symbolic name displayed in the device editor during and after placement. The API qualifies all register names with user assigned instance and block name to provide direct access to the TachTimer registers through the API include files. The block names are TIMER16_LSB and TIMER16_MSB.

Parameters and Resources

Once a TachTimer16 User Module has been selected and placed using the Device Editor, values may be selected and altered for the following parameters.

Clock

The Clock parameter is selected from one of 15 sources. These sources include the VC1, VC2, VC3, other PSoC blocks, and external inputs routed through global inputs and outputs. When using an external digital clock for the block, the row input synchronization should be turned off for best accuracy, and sleep operation.

Capture

This parameter is selected from one of 16 sources. A rising edge on this input causes the Count register to be transferred to the Compare register. The software capture mechanism will not operate correctly if this parameter is set to a value of one or is held high externally.

TerminalCountOut

The terminal count output is an auxiliary TachTimer16 output. This parameter allows it to be disabled or connected to any of the row output busses.

CompareOut

The compare output may be disabled (without interfering with interrupt operations) or connected to any of the row output busses. It is always available as an input to the next higher digital PSoC block and to the analog column clock selection multiplexers, regardless of the setting of this parameter.

Period

This parameter sets the period of the timer. Allowed values are between 0 and 2^{16}-1. This value is loaded into the Period register. The period is automatically reloaded when the counter reaches zero or the timer is enabled from the disabled state. This value may be modified using the API.

CompareValue

This parameter sets the count point in the timer period when a compare event is triggered. This value is loaded into the Compare register. Allowed values are between zero and the period value. This value may be modified using the API.

CompareType

This parameter sets the compare function type **Less Than** or **Less Than or Equal** as described in the functional description, above.

ClockSync

In the PSoC devices, digital blocks may provide clock sources in addition to the system clocks. Digital clock sources may even be chained in ripple fashion. This introduces skew with respect to the system clocks. Appropriate values for this parameter should be determined from the following table.

ClockSync Value	Use
Sync to SysClk	Use this setting for any 24 MHz (SysClk) derived clock source that is divided by two or more. Examples include VC1, VC2, VC3 (when VC3 is driven by SysClk), 32KHz, and digital PSoC blocks with SysClk-based sources. Externally generated clock sources should also use this value to ensure that proper synchronization occurs.
Sync to SysClk*2	Use this setting for any 48 MHz (SysClk*2) based clock unless the resulting frequency is 48 MHz (in other words, when the product of all divisors is 1).
Use SysClk Direct	Use when a 24 MHz (SysClk/1) clock is desired. This does not actually perform synchronization but provides low-skew access to the system clock itself. If selected, this option overrides the setting of the Clock parameter, above. It should always be used instead of VC1, VC2, VC3 or digital Blocks where the net result of all dividers in combination produces a 24 MHz output.
Unsynchronized	Use when the 48 MHz (SysClk*2) input is selected. Use when unsynchronized inputs are desired. In general this use is advisable only when interrupt generation is the sole application of the TachTimer16. This setting is required for blocks that remain active during sleep.

TC_PulseWidth

This parameter provides the means of specifying whether the terminal count output pulse is one clock cycle wide or one half clock cycle wide.

InvertCapture

This parameter determines the sense of the Capture Input signal. When **Normal** is selected, the Capture input is active-high. Selecting **Invert** causes the sense to be interpreted as active-low.

Application Programming Interface

The Application Programming Interface (API) routines are provided as part of the user module to allow the designer to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the “include" files.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the TachTimer_1 to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable and constant symbol. In the following descriptions the instance name has been shortened to TachTimer for simplicity.

Note

In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X prior to the call if those values are required after the call. This “registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy,

too. Though some user module API function may leave A and X unchanged, there is no guarantee they will do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

TachTimer16_PERIOD

Description:

Constant that represents the value chosen for the Period field of the TachTimer16 in the Device Editor. The value can have a range between 0 and 65535.

TachTimer16_COMPARE_VALUE

Description:

Constant that represents the value chose for the PulseWidth field of the TachTimer16 in the Device Editor. The value can have a range between 0 and 65535.

TachTimer16_Start

Description:

Starts the TachTimer16 operation. The Count register will be decremented on the next clock cycle.

C Prototype:

```
void  TachTimer16_Start(void);
```

Assembly:

```
lcall  TachTimer16_Start
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

TachTimer16_Stop

Description:

Stops the TachTimer16 operation.

C Prototype:

```
void  TachTimer16_Stop(void);
```

Assembly:

```
lcall  TachTimer16_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The output will be set low and subsequent writes to the Period register will cause the Count register to update with the new period value. The A and X registers may be altered by this function.

TachTimer16_EnableInt

Description:

Enables the interrupt mode operation for both capture and terminal count interrupts. Note, however, that global interrupts must also be enabled before interrupts will actually be serviced.

C Prototype:

```
void  TachTimer16_EnableInt(void);
```

Assembly:

```
lcall  TachTimer16_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

This routine modifies the appropriate interrupt enable register in IO space. The A and X registers may be altered by this function.

TachTimer16_EnableTerminalInt

Description:

Enables the interrupt mode operation for terminal count interrupt. Note, however, that global interrupts must also be enabled before interrupts will actually be serviced.

C Prototype:

```
void  TachTimer16_EnableTerminalInt(void);
```

Assembly:

```
lcall  TachTimer16_EnableTerminalInt
```

Parameters:

None

Return Value:

None

Side Effects:

This routine modifies the appropriate interrupt enable register in IO space. The A and X registers may be altered by this function.

TachTimer16_EnableCaptureInt

Description:

Enables the interrupt mode operation for capture interrupt. Note, however, that global interrupts must also be enabled before interrupts will actually be serviced.

C Prototype:

```
void  TachTimer16_EnableCaptureInt(void);
```

Assembly:

```
lcall  TachTimer16_EnableCaptureInt
```

Parameters:

None

Return Value:

None

Side Effects:

This routine modifies the appropriate interrupt enable register in IO space. The A and X registers may be altered by this function.

TachTimer16_DisableInt

Description:

Disables the interrupt mode operation for both terminal count and capture interrupts.

C Prototype:

```
void  TachTimer16_DisableInt(void);
```

Assembly:

```
lcall  TachTimer16_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

This routine modifies the appropriate interrupt enable register in IO space. The A and X registers may be altered by this function.

TachTimer16_DisableTerminalInt

Description:

Disables the interrupt mode operation for terminal count interrupt.

C Prototype:

```
void  TachTimer16_DisableTerminalInt(void);
```

Assembly:

```
lcall  TachTimer16_DisableTerminalInt
```

Parameters:

None

Return Value:

None

Side Effects:

This routine modifies the appropriate interrupt enable register in IO space. The A and X registers may be altered by this function.

TachTimer16_DisableCaptureInt

Description:

Disables the interrupt mode operation for capture interrupt.

C Prototype:

```
void  TachTimer16_DisableCaptureInt(void);
```

Assembly:

```
lcall  TachTimer16_DisableCaptureInt
```

Parameters:

None

Return Value:

None

Side Effects:

This routine modifies the appropriate interrupt enable register in IO space. The A and X registers may be altered by this function.

TachTimer16_WritePeriod

Description:

Writes the Period register with the period value. The period will be loaded into the Count register, when the zero-count condition is reached or immediately if the TachTimer16 is currently stopped.

C Prototype:

```
void  TachTimer16_WritePeriod(WORD wPeriod);
```

Assembly:

```
mov   X, [wPeriod]           ; place MSB in X
mov   A, [wPeriod+1]         ; place LSB in A
lcall  Timer16_WritePeriod
```

Parameters:

wPeriod: wPeriod is a value between 0 and 2^{16}-1, to set the TachTimer16 period. MSB is passed in the X register and LSB is passed in the Accumulator.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

TachTimer16_WriteCompareValue

Description:

Modifies the value of the TachTimer16 Compare register. In order to avoid unexpected side effects, the TachTimer16 should be disabled (not yet enabled via the Start API function or by first calling the Stop API function).

C Prototype:

```
void  TachTimer16_WriteCompareValue(WORD wCompareValue);
```

Assembly:

```
mov   X, [wCompareValue]           ; place MSB in X
mov   A, [wCompareValue+1]         ; place LSB in A
lcall  TachTimer16_WriteCompareValue
```

Parameters:

wCompareValue: wCompareValue is a value between 0 and the Period register value, to set the TachTimer16 compare value. MSB is passed in the X register and LSB is passed in the Accumulator.

Return Value:

None

Side Effects:

If this function is called while the TachTimer16 is running and the compare value is equal to or greater than the current value of the Count register, then a compare event can occur. The value of the compare register may vary somewhat unpredictably as the Compare register is distributed across multiple PSoC blocks and written one byte at a time. The order in which the bytes are written is not specified and subject to change. This could cause an interrupt, if both the interrupt type is set to trigger on the compare event and the TachTimer16 interrupt is enabled. The A and X registers may be altered by this function.

TachTimer16_wReadCompareValue

Description:

Reads the TachTimer16 Compare registers.

C Prototype:

```
WORD  TachTimer16_wReadCompareValue(void);
```

Assembly:

```
lcall  TachTimer16_wReadCompareValue
mov   [wCompareValue], X           ; MSB returned in X
mov   [wCompareValue+1], A         ; LSB returned in A
```

Parameters:

None

Return Value:

wCompareValue: Compare register contents. MSB is passed in the X register and LSB is passed in the Accumulator.

Side Effects:

The A and X registers may be altered by this function.

TachTimer16_wReadTimerSaveCV

Description:

Reads the current TachTimer16 Count register value, while preserving the Compare registers. This performs a software-solicited, hardware-synchronous counter capture operation. This function should only be used if the contents of the Compare register must be preserved. If the Compare register contents do not need to be preserved, then using the wReadTimer() function is preferred. Note that this API routine used to be called wReadCounter.

C Prototype:

```
WORD  TachTimer16_wReadTimerSaveCV(void);
```

Assembly:

```
lcall  TachTimer16_wReadTimerSaveCV
mov   [wCount], X           ; MSB returned in X
mov   [wCount+1], A         ; LSB returned in A
```

Parameters:

None

Return Value:

wCount: Count register contents. MSB is passed in the X register and LSB is passed in the Accumulator.

Side Effects:

In order to read the value of the Count register, its value must be momentarily transferred to the Compare register before it can be returned. This causes the compare condition to become true immediately or on the next TachTimer16 input clock cycle depending on whether the CompareType parameter is set to “Less than or Equal to," or “Less Than," respectively. If (or when) the user module and global interrupts are enabled, the interrupt will be serviced, quite possibly before this API function has returned to the caller and even before it has restored the Compare register to its previous state. Interrupts are momentarily disabled. Finally, in order to restore the Compare register, the user module itself is temporarily disabled. This may cause the Count register to miss one or more counts. The A and X registers may be altered by this function.

TachTimer16_wReadTimer

Description:

Reads the current TachTimer16 Count register value, while preserving the Compare registers. This performs a software-solicited, hardware-synchronous counter capture operation. This is the preferred method of reading the Count registers, providing that the Compare registers are not required to be preserved. Note that this API routine used to be called wCaptureCounter.

C Prototype:

```
WORD  TachTimer16_wReadTimer(void);
```

Assembly:

```
lcall  TachTimer16_wReadTimer
mov   [wCount], X           ; MSB returned in X
mov   [wCount+1], A         ; LSB returned in A
```

Parameters:

None

Returns:

wCount: Count register contents. MSB is passed in the X register and LSB is passed in the Accumulator.

Side Effects:

Compare register contents are lost. The compare condition becomes true immediately or on the next TachTimer16 input clock cycle depending on whether the CompareType parameter is set to “Less than or Equal to," or “Less Than," respectively. If (or when) the user module and global interrupts are enabled, the interrupt will be serviced, quite possibly before this API function has returned control to its caller. The A and X registers may be altered by this function.

Sample Firmware Source Code

The following sections of sample code assume that the TachTimer16 User Module is named "TachTimer16". The sample code written in assembly language is a subroutine that configures the TachTimer16 to wait for a capture event within a time window. Either the capture event occurs or the time window elapses. This sample code implementation uses the TachTimer16's interrupts. Therefore, ISR code is added in the TachTimer16INT.asm file..

The following code is assembly language source that illustrates the use of some of the APIs:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Description:
; This sample shows how to capture an event with a bounded time limit
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
include "M8C.inc" ; include the global include file
include "psocapi.inc" ; include the API include file

area bss (RAM,REL)
_wElapsedTime:: BLK 2                ; Word variable for the amount of elapsed time
_fOverflow:: BLK 1                   ; Flag variable to check for overflow

area text (ROM,REL)
_main::
    call CapturePulse                ; Call CapturePulse routine

.terminate:
    jmp .terminate

CapturePulse::
    mov [_wElapsedTime+0], 0         ; Zero out all variables
    mov [_wElapsedTime+1], 0
    mov [_fOverflow], 0

  mov A, FFh
  mov X, FFh
    call TachTimer16_WritePeriod     ; Set the period to the maximum
```

```
    mov X, 0
  mov A, 0
    call TachTimer16_WriteCompareValue ; Set the compare to trigger at 0
   call TachTimer16_EnableInt       ; Enable the Capture and Terminal Count interrupts
    M8C_EnableGInt                    ; Enable global interrupts
    call TachTimer16_Start            ; Start the timer

.WaitForCaptureOrOverflow:
    mov A, [_wElapsedTime+0]
  or A, [_wElapsedTime+1]           ; Check if wElapsedTime is still zero
    or A, [_fOverflow]                ; Check if fOverflow is still zero
    jz .WaitForCaptureOrOverflow      ; Keep checking if capture or overflow still
have not occurred
    cmp [_fOverflow], 0               ; Check if event was an overflow
    jz .TimerDone                     ; Jump to TimerDone if it was not overflow
.Overflow:
;Timer overflow occurred. Capture event did not occur within timer
;period. This may indicate absence or too low frequency of input
;signal.
    ret
.TimerDone:
;Evaluate captured value here!
; return to caller when complete
    ret
```

The Capture interrupt routine, located in the file *capture_int.asm*, is:

```
_TachTimer16_LSB_ISR:

   ;@PSoC_UserCode_BODY_LSB@ (Do not change this line.)

   ;---------------------------------------------------
   ; Insert your custom code below this banner
   ;---------------------------------------------------
push X; Save X and A values on stack
push A
call TachTimer16_wReadTimer; Read the captured timer value
mov [_wElapsedTime+1], A; Move the value into wElapsedTime
mov [_wElapsedTime], X
call TachTimer16_Stop; Stop the timer
pop A; Restore X and A
pop X

   ;---------------------------------------------------
   ; Insert your custom code above this banner
   ;---------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)

   reti
```

The TerminalCount interrupt routine, located in the file *capture_int.asm*, is as follows.

```
_TachTimer16_MSB_ISR:

   ;@PSoC_UserCode_BODY_MSB@ (Do not change this line.)
```

```
   ;-----------------------------------------------
   ; Insert your custom code below this banner
   ;-----------------------------------------------
 push X; Save X and A on stack since
                              ; TachTimer16_Stop may modify them
push A
mov [_fOverflow], 1   ; Set fOverflow to non-zero value
call TachTimer16_Stop ; Stop the timer
pop A; Restore X and A
pop X
   ;-----------------------------------------------
   ; Insert your custom code above this banner
   ;-----------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)

   reti
```

The same code in C is as follows. Note that the interrupt routine must be written in assembly.

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules


WORD wElapsedTime;
BYTE fOverflow;

void CapturePulse( void )
{
    TachTimer16_WritePeriod(0xffff);        // Set the period to the maximum
    TachTimer16_WriteCompareValue(0x0000);  // Set the compare value to 0
    TachTimer16_EnableInt();                // Enable capture and terminal
                                            // counter interrupts
    M8C_EnableGInt;                         // Enable global interrupts
    TachTimer16_Start();                    // Start the timer
    while((wElapsedTime == 0) && (fOverflow == 0));    // Wait here for
                                            // capture or overflow events
    if(fOverflow == 0)                      // Check if the event
                                            // was an overflow event
    {
        /* Process the data here */
    } else {
        /* Place Timer Overflow processing code here */
    }
}

void main(void)
{
    CapturePulse(); // Call CapturePulse function from main
}
```

Configuration Registers

The 16-bit TachTimer16 uses two digital PSoC blocks. In placement order from left to right they are named TIMER16_LSB and TIMER16_MSB. Each block is personalized and parameterized through 7 registers. The following tables give the “personality" values as constants and the parameters as named bitfields with brief descriptions. Symbolic names for these registers are defined in the user module instance’s C and assembly language interface files (the “.h" and “.inc" files).

Table 2. Function Register (DxBxxFN), Bank 1

Block/Bit	7	6	5	4	3	2	1	0
MSB	0	0	1	Compare	0	0	0	0
LSB	InvertCapture	BCEN	0	Compare	0	0	0	0

BCEN gates the terminal count output onto the row broadcast bus line. This bitfield is set in the Device Editor by directly configuring the broadcast line. The InputInvert flag, set through a user module parameter displayed in the Device Editor, controls the sense of the capture input signal. The Compare flag indicates whether the compare function is set to **Less Than or Equal To** or **Less Than**. The Compare is set in the Device Editor directly through user module parameters described in the earlier section on the topic.

Table 3. Input Register (DxBxxIN), Bank 1

Block/Bit	7	6	5	4	3	2	1	0
MSB	0	0	1	1	Clock			
LSB	Capture				Clock			

Capture selects the input signal of the same name from one of 16 sources. The User Module “Capture" parameter setting in the Device Editor determines its value. Similarly, the user module “Clock" parameter setting determines this value.

Table 4. Output Register (DxBxxOU), Bank 1

Block/Bit	7	6	5	4	3	2	1	0
MSB	ClockSync		AuxEnable	AuxSelect		OutEnable	OutputSelect	
LSB	ClockSync		0	0	0	0	0	0

The user module “ClockSync" parameter in the Device Editor determines the value of the ClockSync bits in both register. Though similarly named, the AuxEnable and AuxSelect bits are related, instead, to the OutEnable and OutSelect bitfields. AuxEnable and AuxSelect permit driving the compare output signal onto one of the row output busses and are controlled by manipulating the row bus graphically in the Device Editor Interconnect View. OutEnable is set when the terminal count output is driven onto one of the row or global output busses. OutputSelect controls which of the busses will be driven from the compare output.

Table 5. Control Register (DxBxxCR0), Bank 0

Block/Bit	7	6	5	4	3	2	1	0
MSB	0	0	0	0	0	TC_PulseWidth	0	0
LSB	0	0	0	0	0	0	1	Start/Stop

The TC_PulseWidth is controlled by user module parameter with the same name and provides the means of specifying whether the terminal count output pulse is one clock cycle wide or one half clock cycle wide. Start/Stop indicates that the TachTimer16 is enabled when set and disabled when clear. It is modified by using the TachTimer16 API.

Table 6. Count Register (DxBxxDR0), Bank 0

Block/Bit	7	6	5	4	3	2	1	0
MSB	Count (MSB)							
LSB	Count (LSB)							

The Count register is the 16-bit down count value decremented by 1 in every clock cycle that the enable input is active. Its value is loaded from the contents of the Period register in the clock cycle following the terminal count (zero value). It can be read using the TachTimer16 API.

Table 7. Period Register (DxBxxDR1), Bank 0

Block/Bit	7	6	5	4	3	2	1	0
MSB	Period (MSB)							
LSB	Period (LSB)							

The Period register is a write-only register that can be set through the Device Editor and by the TachTimer16 API. When written, the value is transferred to the Count register if the user module is disabled through the API. Its value is automatically copied into the Count register in the clock cycle following terminal count.

Table 8. Compare Register (DxBxxDR2), Bank 0

Block/Bit	7	6	5	4	3	2	1	0
MSB	Compare (MSB)							
LSB	Compare (LSB)							

The Compare register holds the value against which the Count register is tested in order to generate the compare output. It can be set by the Device Editor and the TachTimer16 API.

Version History

Version	Originator	Description
1.1	TDU	Updated Clock description to include: When using an external digital clock for the block, the row input synchronization should be turned off for best accuracy, and sleep operation.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2008-2014 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.