Experiment 10

Timers and Time Base Generation

Timers/Counters

Almost every microcontroller comes with one or more (sometimes many more) built-in timer/-counters, and these are extremely useful to the embedded programmer. The term timer/counter itself reflects the fact that the underlying counter hardware can usually be configured to count either regular clock pulses (making it a timer) or irregular event pulses (making it a counter). Sometimes timers are also called "hardware timers" to distinguish them from software timers which are bits of software that perform some timing function.

What is a Timer?

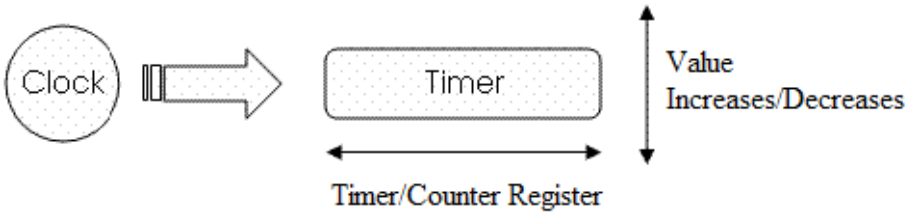
We use timers every day - the simplest one can be found on your wrist. A simple clock will time the seconds, minutes and hours elapsed in a given day - or in the case of a twelve hour clock, since the last half-day. ARM timers do a similar job, measuring a given time interval.

Micro-controllers, such as the LM4F120 utilize hardware timers to generate signals of various frequencies, generate pulse-width-modulated (PWM) outputs, measure input pulses, and trigger events at known frequencies or delays. The LM4F120 parts have several different types of timer peripherals which vary in their configurability. The simplest timers are primarily limited to generating signals of a known frequency or pulses of fixed width. While more sophisticated timers add additional hardware to utilize such a generated frequency to independently generate signals with specific pulse widths or measure such signals.

How a Timer Works?

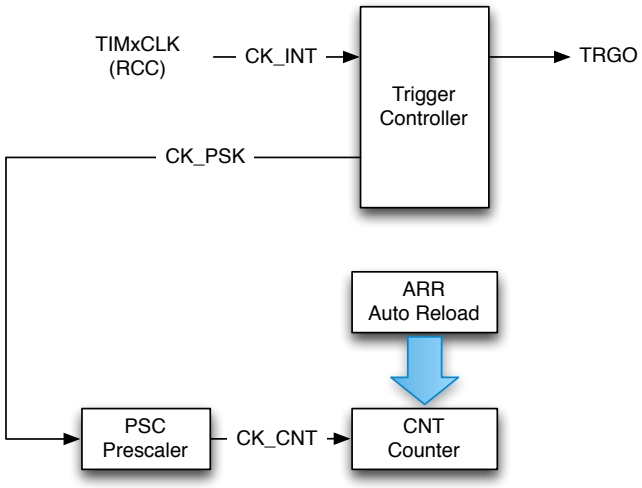
An ARM timer in simplest term is a register. Timers generally have a resolution of 16/32 or 32/64 bits. So a 16 bit timer is 16 bits wide, and is capable of holding value within 0-65535. But this register has a magical property - its value increases/decreases automatically at a predefined rate (supplied by user). This is the timer clock and this operation does not need CPUs intervention.

An example of a basic timer is illustrated in Figure 10.2. This timer has four components a controller, a prescaler (PSC), an "auto-reload" register (ARR) and a counter (CNT). The function of the prescaler is to divide a reference clock to lower frequency. The LM4F120 timers



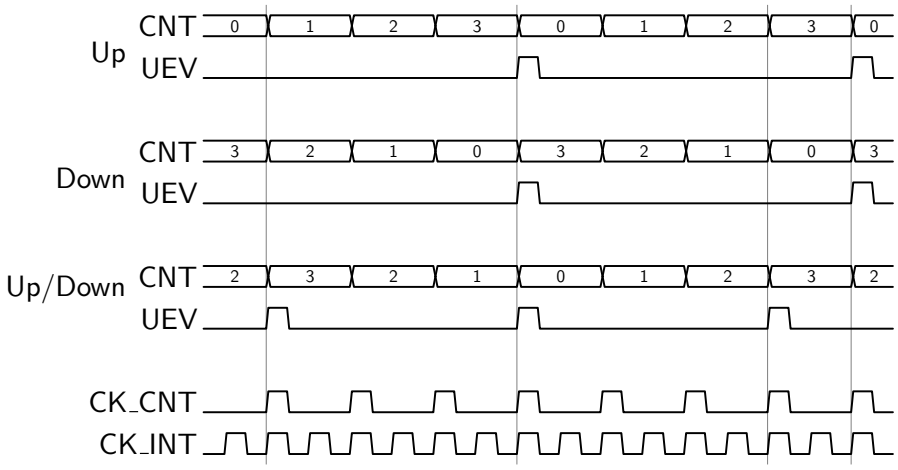
Timer Register

have 8-bit prescaler registers for 16-bit timers which can divide the reference clock by any value 1..255 and 16-bit prescalar for 32-bit timer which can divide the reference clock by any value 1..65535. The counter register can be configured to count up, down, or up/down and to be reloaded from the auto reload register whenever it wraps around (an "update event") or to stop when it wraps around. The basic timer generates an output event (TGRO) which can be configured to occur on an update event or when the counter is enabled (for example on a GPIO input).



Basic Timer

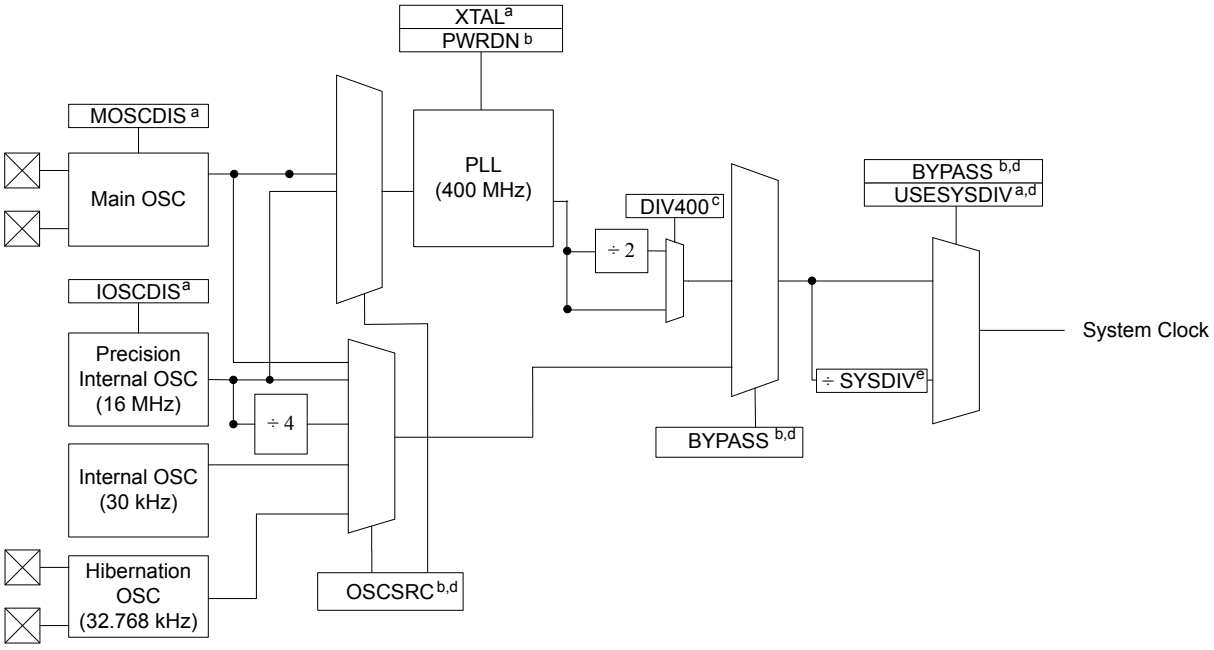
To understand the three counter modes consider Figure 10.3. In these examples, we assume a prescaler of 1 (counter clock is half the internal clock), and a auto reload value of 3. Notice that in "Up" mode, the counter increments from 0 to 3 (ARR) and then is reset to 0. When the reset occurs, an "update event" is generated. This update event may be tied to TRGO, or in more complex timers with capture/compare channels it may have additional effects. Similarly, in "Down" mode, the counter decrements from 3 to 0 and then is reset to 3 (ARR). In Down mode, an update "event" (UEV) is generated when the counter is reset to ARR. Finally, in Up/Down mode, the counter increments to ARR, then decrements to 0, and repeats. A UEV is generated before each reversal with the effect that the period in Up/Down mode is one shorter than in either Up or Down mode.



Counter Modes (ARR=3, PSC=1)

Clock Configuration

The clock system on the Stellaris Launchpad is extremely flexible. The clock tree of system clock configuration is shown in Figure 10.4



System Clock Tree

The internal system clock (SysClk), is derived from any of the four reference sources plus two others: the output of the main internal PLL and the precision internal oscillator divided by four (4 MHz 1%). The frequency of the PLL clock reference must be in the range of 5 MHz to 25 MHz (inclusive). Following table shows how the various clock sources can be used in a system.

Clock Source	Drive PLL?		Used as SysClk?	
Precision Internal Oscillator	Yes	BYPASS = 0, OSCSRC = 0x1	Yes	BYPASS = 1, OSCSRC = 0x1
Precision Internal Oscillator divide by 4 (4 MHz ± 1%)	No	-	Yes	BYPASS = 1, OSCSRC = 0x2
Main Oscillator	Yes	BYPASS = 0, OSCSRC = 0x0	Yes	BYPASS = 1, OSCSRC = 0x0
Low-Frequency Internal Oscillator (LFIOSC)	No	-	Yes	BYPASS = 1, OSCSRC = 0x3
Hibernation Module 32.768-kHz Oscillator	No	-	Yes	BYPASS = 1, OSCSRC2 = 0x7

Clock Source Options

The Run-Mode Clock Configuration (RCC) and Run-Mode Clock Configuration 2 (RCC2) registers provide control for the system clock. The RCC2 register is provided to extend fields that offer additional encodings over the RCC register. When used, the RCC2 register field values are used by the logic over the corresponding field in the RCC register. In particular, RCC2 provides for a larger assortment of clock configuration options. These registers control the following clock functionality:

- Source of clocks in sleep and deep-sleep modes
- System clock derived from PLL or other clock source
- Enabling/disabling of oscillators and PLL
- Clock divisors
- Crystal input selection

To configure RCC and RCC2 clock configuration registers consult the data sheet of LM4F120.

Timers in LM4F120

The Stellaris General-Purpose Timer Module (GPTM) in LM4F120 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 or event counters. Complete register map of the GPTM is given in the data sheet [pg. 684]. Some key registers are described below:

SYSCTL_RCGCTIMER_R - enable clock for GPTM (enable clock for Timer0)

TIMER0_CTL_R - control Timer0 module

TIMER0_CFG_R - control global operation of Timer0 module (use 16/32-bit mode)

TIMER0_TAMR_R - control the mode for Timer0

TIMER0_TAPR_R - set the prescalar for Timer0

TIMER0_TAILR_R - load the starting count value into the timer when counting down and set the upper bound for the timeout event when counting up

How to Configure a Timer for Periodic Interrupts

The GPTM is configured for Periodic mode by the following sequence:

1. Enable the General Purpose Timer Module (GPTM) by asserting the appropriate TIMERn bit in the RCGCTIMER register. [pg. 312]
2. Ensure the timer is disabled (TnEN bit in the GPTMCTL register is cleared for the corresponding timer) before making any changes. [pg. 696]
3. Write the GPTM Configuration Register (GPTMCFG) with a value of 0x04 to configure in 16-bit mode. [pg. 686]
4. Write a value of 0x2 in the TnMR field of the GPTM Timer n Mode Register (GPTMTnMR) to configure it in periodic mode. [pg. 688]
5. Load the period value into the GPTM Timer n Interval Load Register (GPTMTnILR). [pg. 715]
6. Load the prescale value into the GPTM Timer n Prescale Register (GPTMTnPR). [pg. 719]
7. Assert Timer A Time-Out Clear Interrupt (TATOCINT) bit of GPTM Interrupt Clear Register (GPTMICR) to clear the time-out flag. [pg. 713]
8. Set Timer A Time-Out Interrupt Mask (TATOIM) bit of GPTM Interrupt Mask Register (GPTMIMR) to enable the time-out interrupt. [pg. 704]
9. Set the priority in the corresponding NVIC Priority Register. [Table 2.9, pg. 101], [pg. 149]
10. Enable the correct interrupt in the corresponding NVIC Enable Register. [pg. 139]
11. Set the TnEN bit in the GPTMCTL register to enable the timer and start counting. [pg. 696]

Source Code

Complete source code for generating the periodic interrupts for Timer 0A (configured in 16-bits). This program toggles the state of PF1 with 1 Hz frequency.

Example 10.1: timer.h

```
// GPIO registers
#define SYSCTL_RCGCGPIO_R       (*((volatile unsigned long *)0x400FE608))
#define GPIO_PORTF_DATA_R       (*((volatile unsigned long *)0x400253FC))
#define GPIO_PORTF_DIR_R        (*((volatile unsigned long *)0x40025400))
#define GPIO_PORTF_DEN_R        (*((volatile unsigned long *)0x4002551C))

// Timer registers
#define SYSCTL_RCGCTIMER_R      (*((volatile unsigned long *)0x400FE604))
```

```
#define TIMER0_CTL_R            (*((volatile unsigned long *)0x4003000C))
#define TIMER0_CFG_R            (*((volatile unsigned long *)0x40030000))
#define TIMER0_TAMR_R           (*((volatile unsigned long *)0x40030004))
#define TIMER0_TAILR_R          (*((volatile unsigned long *)0x40030028))
#define TIMER0_TAPR_R           (*((volatile unsigned long *)0x40030038))
#define TIMER0_ICR_R            (*((volatile unsigned long *)0x40030024))
#define TIMER0_IMR_R            (*((volatile unsigned long *)0x40030018))

// NVIC registers
#define NVIC_EN0_R              (*((volatile unsigned long *)0xE000E100))
#define NVIC_PRI4_R             (*((volatile unsigned long *)0xE000E410))

// constant values
#define TIM0_CLK_EN             0x_____  // enable clock for Timer0
#define TIM0_EN                 0x_____  // disable Timer0 before setup
#define TIM_16_BIT_EN           0x_____  // configure 16-bit timer mode
#define TIM_TAMR_PERIODIC_EN    0x_____  // configure periodic mode
#define TIM_FREQ_10usec         0x_____  // select prescalaer for
                                         // desired frequency 100 kHz
#define TIM0_INT_CLR            0x_____  // clear time out interrupt
#define EN0_INT19               0x_____  // enable interrupt 19
#define PORTF_CLK_EN            0x_____  // enable clock for port F
#define TOGGLE_PF1              0x_____  // toggle red led (PF1)
#define LED_RED                 0x_____  // configure red led (PF1)

// function headers
void GPIO_Init(void);
void Timer_Init(unsigned long period);
void DisableInterrupts(void);
void EnableInterrupts(void);
void WaitForInterrupt(void);
```

Example 10.2: Timer0A_Periodic_Interrupt.c

```
#include "timers.h"

void Timer_Init(unsigned long period){

  SYSCTL_RCGCTIMER_R |= TIM0_CLK_EN;      // enable clock for Timer0
  TIMER0_CTL_R &= ~(TIM0_EN);             // disable Timer0 before setup
  TIMER0_CFG_R |= TIM_16_BIT_EN;          // configure 16-bit timer mode
  TIMER0_TAMR_R |= TIM_TAMR_PERIODIC_EN;  // configure periodic mode
  TIMER0_TAILR_R = period;                // set initial load value
  TIMER0_TAPR_R = TIM_FREQ_10usec;        // set prescalaer for desired
      frequency 100 kHz
  TIMER0_ICR_R = TIM0_INT_CLR;            // clear time out interrupt
    TIMER0_IMR_R |= TIM0_EN;                // enable interrupt mask for
```

```
        Timer_0A

  DisableInterrupts();
  // Set priority for interrupt
  NVIC_PRI4_R = (NVIC_PRI4_R & 0x00FFFFFF)|0x40000000;
  NVIC_EN0_R |= EN0_INT19;                 // enable interrupt 19

    TIMER0_CTL_R |= TIM0_EN;                 // enable Timer_0A
  EnableInterrupts();
}

void GPIO_Init(){

  SYSCTL_RCGCGPIO_R |= PORTF_CLK_EN;
  GPIO_PORTF_DIR_R |= LED_RED;
  GPIO_PORTF_DEN_R |= LED_RED;
}

void Timer0A_Handler(void){
  TIMER0_ICR_R = TIM0_INT_CLR;
  GPIO_PORTF_DATA_R ^= TOGGLE_PF1;

}

int main(void){
  Timer_Init(50000);                  // generate a square wave for 2 Hz
  GPIO_Init();                        // initialize PF1 as digital output
  while(1);
}
```