Timers
The ARM architecture features four timer/counters; two of these are 16-bit counters and two are 32-bit counters. The counters are named TMR16B0, TMR16B1, TMR32B0, and TMR32B1. The general characteristics of the four timers are:
• Each timer has a corresponding prescaler. The prescaler allows the user to divide the clock rate coming into the timer.
• Can capture timer value on an external input trigger.
• The match registers that allow:
  - Continuous operation with optional interrupt generation on match.
  - Stop timer on match with optional interrupt generation.
  - Reset timer on match with optional interrupt generation.
• External outputs corresponding to match registers, with the following capabilities:
  - Set low on match.
  - Set high on match.
  - Toggle on match.
  - Do nothing on match.

All of the timers are driven by the Peripheral Clock. The peripheral clock is derived from the crystal clock which is the processor clock. Figure 1 shows a block diagram of the system clock. The line labeled "AHB clocks 1 to 18 (memories and peripherals)" contains the peripheral clock for the timers.

Figure 1
System clock diagram.
Four bits in the SYSAHBCLKCNTL registers enable/disable the clock for the timers. These four bits are bits 7 (16-bit Timer 0), 8 (16-bit Timer 1), 9 (32-bit Timer 0), and 10 (32-bit Timer 1). A 0 disables the timer and a 1 enables it.

Figure 2 shows a block diagram for a typical timer/counter. The peripheral clock goes into the timer prescale counter. This counter can be used to reduce the frequency of the peripheral clock for a particular timer.

In our case, the crystal clock runs at 48MHz. The clock is driven at the crystal frequency divided by the prescaler value plus 1. For example, if the prescaler is set to 0, the timer will run at 48MHz/(0+1) = 48MHz. etc. In C-code we can write:

```c
TMR16B1PR = 3; //Prescale register. Divide 48 MHz/(3+1) = 12 MHz
```

The timers for this processor do not overflow to generate an interrupt. Instead, each timer has four associated match register (MR0 to MR3). Each match register can be set with a value which
is compared to the timer on each timer tick. If the match register is the same as the timer, you can so the following operations:

• Stop timer on match and optionally generate an interrupt.
• Reset timer on match and optionally generate an interrupt.
• Set a bit low on match.
• Set a bit high on match.
• Toggle a bit on match.
• Do nothing on match.

To use a timer in the polled mode we need to do the following:

1. Enable the appropriate timer in SYSAHBCLKCTRL.
2. Load the prescale register with the clock divisor. Remember that the clock is divided by the prescale value plus 1.
3. Load the match register with the number of clock ticks needed.
4. Set the match control register so that the match register generates an interrupt when a match occurs.
5. Set the match register to reset the timer when a match occurs.
6. Set the timer control register to start the timer running.
7. When the timer hits a match a bit in the timer interrupt register will be set to 1. Write a while loop to wait until this happens.
8. Do whatever it is you need to when the time has passed.
9. Reset the interrupt flag by writing a 1 (not a zero) to the interrupt register.

Example 1 on the following page shows how to use the 16-bit timer 1 to toggle P0.1 every 42.67 μsec.
Example 1

Toggles P0.1 every 42.67 μsec using a polled timer.

```c
//TimerPolled.c
/* This program toggles bit P0.1 based on the 16-bit timer 1 and match
 register 0. The timer is polled. Interrupts are turned off but
 the match register is set to produce an interrupt on match. This
 interrupt bit is polled to determine when to toggle the bit.
 */
int main()
{
    SYSAHBCLKCTRL |= (1 << 8);  //Enable clock for 16-bit timer 1
    GPIO0DIR = 2;                //Port0.1 set to output
    TMR16B1PR = 1;               //Prescale register. Divide P Clock by 2 = 24 MHz
    //MR0 sets the base frequency. Clock period is 1/24MHz = 0.04167usec.
    // 1024 x 0.04167usec = 42.67 usec = 23.435 KHz base frequency
    TMR16B1MR0 = 1024;           //Match count is 1024 -> generates interrupt
    TMR16B1MCR |= 1;             //Generate an interrupt if match on MR0
    TMR16B1MCR |= (1 << 1);      //Causes TimerCounter to be reset if match on MR0
    TMR16B1TCR |= 1;             //Enable TimerCounter to run
    while(1)
    {
        while((TMR16B1IR & 0x01) == 0); //wait for timer to reach count
        GPIO0DATA ^= 0x2;            //Toggle P0.1
        TMR16B1IR = 1;              //Reset interrupt bit
    }
}
```
Interrupts and Exceptions

In general an interrupt is triggered by internal or external hardware which requires service. An exception is triggered by a fault such as a power failure, execution of an unknown opcode, illegal bus or memory operation, etc. Figure 3 lists all of the exceptions and interrupts on the ARM Cortex M0 processor. Exceptions are given negative IRQ numbers and interrupts are given positive IRQ numbers. Each exception or interrupt has a 4 byte space in a vector table where the service routine for the interrupt or exception is stored. Figure 4 shows a diagram of the vector table. In general IRQn has a vector stored at 0x40 + 4n.

When an interrupt or an exception occurs the processor branches to the address stored in the vector table. For example if interrupt IRQ1 occurs the processor will go to location 0x44 in memory to find the address of the service routine for that interrupt.

To create an interrupt function in C-code you need to do four things: 1) Set up the vector table 2) Set the priority level 3) Define the interrupt service routine 4) Enable the interrupt. The vector table gets set up and defined for you by the compiler when it finds your interrupt service routine. You can also use the default priority level unless you have multiple interrupts that may overlap.

### Table 2.11. Properties of the different exception types

<table>
<thead>
<tr>
<th>Exception number</th>
<th>IRQ number</th>
<th>Exception type</th>
<th>Priority</th>
<th>Vector address</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>Reset</td>
<td>-3, the highest</td>
<td>0x00000004</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>2</td>
<td>-14</td>
<td>NMI</td>
<td>-2</td>
<td>0x00000008</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>3</td>
<td>-13</td>
<td>HardFault</td>
<td>-1</td>
<td>0x0000000C</td>
<td>Synchronous</td>
</tr>
<tr>
<td>4-10</td>
<td>-</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-5</td>
<td>SVC</td>
<td>Call</td>
<td>Configurable</td>
<td>0x0000002C</td>
</tr>
<tr>
<td>12-13</td>
<td>-</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-2</td>
<td>PendSV</td>
<td>Configurable</td>
<td>0x00000038</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>15</td>
<td>-1</td>
<td>SysTick</td>
<td>Configurable</td>
<td>0x0000003C</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exception number [a]</td>
<td>IRQ number [a]</td>
<td>Exception type</td>
<td>Priority</td>
<td>Vector address [b]</td>
<td>Activation</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------</td>
<td>-------------------</td>
<td>------------</td>
</tr>
<tr>
<td>16 and above [d]</td>
<td>0 and above</td>
<td>IRQ</td>
<td>Configurable [e]</td>
<td>0x00000040 and above [f]</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

[a] To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see Interrupt Program Status Register.
[b] See Vector table for more information.
[c] If your device does not implement the SysTick timer, exception number 15 is reserved.
[d] The number of IRQ interrupts is implementation-defined, in the range 1-32. Unimplemented IRQ exception numbers are reserved, for example if the device implements only one IRQ, exception numbers 17 and above are reserved.
[e] See interrupt priority registers on page 4-7
[f] Increasing in steps of 4

Figure 3
Exceptions and interrupts for the ARM Cortex M0 Processor

Figure 4
6-6
With a few modifications we can change the polled timer to one that runs from an interrupt. For the polled timer the interrupt was set up but was never enabled. To enable the interrupt we need to set a bit in the Interrupt Service Enable Register (ISER) like this

\[
\text{NVIC\_ISER} \mid= (1 \ll 17); \quad \text{//Enable 16-bit timer 1 interrupt}
\]

We also need to write the interrupt service routine which must have the name `TIMER16_1_IRQHandler`. It is not necessary to locate the interrupt vector since this is done by the compiler when it finds the interrupt service routine.

Example 2 shows the complete code for the timer interrupt program.
Example 2

```c
#define SYSAHBCLKCTRL (*((volatile unsigned long *) 0x40048080))   //System AHB clock control
#define TMR16B1IR (*((volatile unsigned long *) 0x40010000))   //Timer 1 Interrupt register
#define TMR16B1TCR (*((volatile unsigned long *) 0x40010004))   //Timer 1 Timer control register
#define TMR16B1IP (*((volatile unsigned long *) 0x40010010))   //Timer 1 Interrupt register
#define TMR16B1PC (*((volatile unsigned long *) 0x40010014))   //Timer 1 Prescaler counter register
#define TMR16B1MCR (*((volatile unsigned long *) 0x40010018))   //Timer 1 Match Control register
#define TMR16B1MR0 (*((volatile unsigned long *) 0x40010020))   //Timer 1 Match register 0
#define TMR16B1MR1 (*((volatile unsigned long *) 0x40010024))   //Timer 1 Match register 1
#define TMR16B1MR2 (*((volatile unsigned long *) 0x40010028))   //Timer 1 Match register 2
#define TMR16B1MR3 (*((volatile unsigned long *) 0x4001002C))   //Timer 1 Match register 3
#define TMR16B1CCR (*((volatile unsigned long *) 0x40010030))   //Timer 1 Capture Control register
#define TMR16B1IR (*((volatile unsigned long *) 0x40010034))   //Timer 1 Capture register 0
#define NVIC_ISER (*((volatile unsigned long *) 0xE000E100))  //Int Set Enable Reg
#define GPIO0DATA (*((volatile unsigned long *) 0x50003FFC))   //Port 0 data register
#define GPIO0DIR (*((volatile unsigned long *) 0x50008000))   //Port 0 data direction register
#define GPIO1DATA (*((volatile unsigned long *) 0x50013FFC))   //Port 1 data register
#define GPIO1DIR (*((volatile unsigned long *) 0x50018000))   //Port 1 data direction register
#define SYSAHBCLKCTRL (*((volatile unsigned long *) 0x40048080))   //System AHB clock control

//TimerInt.c

/* This program toggles port 0 bit 1 each time an interrupt is generated 
   by the 16-bit timer 1. The timer clock is set to 24 MHz and 
   match register 0 is set to 1024. When the timer gets to the match 
   value the timer is reset and an interrupt is generated. The ISR 
   turns off the interrupt and toggles P0.1. */

int count;
int main()
{
    SYSAHBCLKCTRL |= (1 << 8);  //Enable clock for 16-bit timer 1
    GPIO0DIR |= 2;            //Port0.1 set to output
    NVIC_ISER |= (1 << 17);  //Enable 16-bit timer 1 interrupt
    TMR16B1IPR = 1;           //Prescaler register. Divide P Clock by 2 = 24 MHz
    TMR16B1MR0 = 1024;       //Match count is 1024 -> generates interrupt
    TMR16B1MCR |= 1;         //Generate an interrupt if match on MR0
    TMR16B1MCR |= (1 << 1);  //Causes TimerCounter to be reset if match on MR0
    TMR16B1TCR |= 1;         //Enable TimerCounter to run
    count = 0;
    while (1);
}

void TIMER16_1_IRQHandler(void)
{
    count++;
    GPIO0DATA ^= 0x2;
    TMR16B1IR = 1;       //clear interrupt
}
```

6-8
Class Exercise 6-1: Rewrite Example 1 to generate a square wave on P0.7 at 500 KHz.

Class Exercise 6-2: Rewrite Example 2 to generate a square wave on P0.7 at 500 KHz.