Week 4
TIMERS for MSP430

Hacettepe University
Loops are OK up to a point but timers are more precise and leave the CPU free for more productive activities. Alternatively, the device can be put into a low-power mode if there is nothing else to be done.

- **Watchdog timer**: Included in all devices (newer ones have the enhanced watchdog timer+). Its main function is to protect the system against malfunctions but it can instead be used as an interval timer if this protection is not needed.

- **Basic timer1**: Present in the MSP430x4xx family only. It provides the clock for the LCD and acts as an interval timer. Newer devices have the LCD_A controller, which contains its own clock generator and frees the basic timer from this task.

- **Real-time clock**: In which the basic timer has been extended to provide a real-time clock in the most recent MSP430x4xx devices.

- **Timer_A**: Provided in all devices. It typically has three channels and is much more versatile than the simpler timers just listed. Timer_A can handle external inputs and outputs directly to measure frequency, timestamp inputs, and drive outputs at precisely specified times, either once or periodically. There are internal connections to other modules so that it can measure the duration of a signal from the comparator, for instance. It can also generate interrupts.

- **Timer_B**: Included in larger devices of all families. It is similar to Timer_A with some extensions that make it more suitable for driving outputs such as pulse-width modulation. Against this, it lacks a feature of sampling inputs in Timer_A that is useful in communication.
Watchdog Timer

- The main purpose of the watchdog timer is to protect the system against failure of the software, such as the program becoming trapped in an unintended, infinite loop.
- Left to itself, the watchdog counts up and resets the MSP430 when it reaches its limit. The code must therefore keep clearing the counter before the limit is reached to prevent a reset.
The operation of the watchdog is controlled by the 16-bit register WDTCTL. It is guarded against accidental writes by requiring the password WDTPW = 0x5A in the upper byte.

A reset will occur if a value with an incorrect password is written to WDTCTL. This can be done deliberately if you need to reset the chip from software.

Reading WDTCTL returns 0x69 in the upper byte, so reading WDTCTL and writing the value back violates the password and causes a reset.
The lower byte of WDTCTL contains the bits that control the operation of the watchdog timer.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>WDT-HOLD</td>
<td>WDT-NMIES</td>
<td>WDTNMI</td>
<td>WDT-TMSEL</td>
<td>WDT-CNTCL</td>
<td>WDTSESEL</td>
<td>WDTISx</td>
<td></td>
</tr>
<tr>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>r0(w)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td></td>
</tr>
</tbody>
</table>

The watchdog is always active after the MSP430 has been reset. By default, the clock is SMCLK, which is in turn derived from the DCO at about 1 MHz. The default period of the watchdog is the maximum value of 32,768 counts, which is therefore around 32 ms. You must clear, stop, or reconfigure the watchdog before this time has elapsed. Stopping the watchdog, means setting the WDTHOLD bit.
The watchdog counter is a 16-bit register WDTCNT, which is not visible to the user.

It is clocked from either SMCLK (default) or ACLK, according to the WDTSEL bit.

The period is 64, 512, 8192, or 32,768 (default) times the period of the clock. This is controlled by the WDTISx bits in WDTCTL. The intervals are roughly 2ms, 16ms, 250ms, and 1000 ms if the watchdog runs from ACLK at 32 KHz.

WDTHOLD = 1 when the WDT+ is not in use conserves power.
If the watchdog is left running, the counter must be repeatedly cleared to prevent it counting up as far as its limit. This is done by setting the WDTCNTCL bit in WDTCTL.

As a result the watchdog timer sets the WDTIFG flag in the special function register IFG1.
Example watchdog application

- The clock is selected from ACLK (WDTSEL = 1) and the longest period (WDTISx = 00), which gives 1s with a 32 KHz crystal for ACLK. It is wise to restart any timer whenever its configuration is changed so the counter is cleared by setting the WDTCNTCL bit. LED1 shows the state of button B1 and LED2 shows WDTIFG.

- The watchdog is serviced by rewriting the configuration value in a loop while button B1 is held down. If the button is left up for more than 1s the watchdog times out, raises the flag WDTIFG, and resets the device with a PUC.

- This is shown by LED2 lighting.
Listing 8.1: Program wdtest1.c to demonstrate the watchdog timer.

// wdtest1.c - trivial program to demonstrate watchdog timer
// Olimex 1121 STK board , 32KHz ACLK
// -----------------------------------------------------------------------------
#include <io430x11x1.h> // Specific device
// -----------------------------------------------------------------------------

// Pins for LEDs and button
#define LED1 P2OUT_bit.P2OUT_3
#define LED2 P2OUT_bit.P2OUT_4
#define B1 P2IN_bit.P2IN_1

// Watchdog config: active, ACLK /32768 -> 1s interval; clear counter
#define WDTCONFIG (WDTCNTCL|WDTSSEL) // Include settings for _RST/NMI pin here as well
// Setting WDTCNTCL = 1 clears the count value to 0000h.

void main (void)
{
    // WDTPW = 0x5A00 or 01011010 00000000
    // WDTSSEL Bit 2 Watchdog timer+ clock source select
    WDTCTL = WDTPW | WDTCONFIG; // Configure and clear watchdog
    P2DIR = BIT3 | BIT4; // Set pins with LEDs to output
    P2OUT = BIT3 | BIT4; // LEDs off (active low)

    for (; ; ) { // Loop forever
        LED2 = ~IFG1_bit.WDTIFG; // LED2 shows state of WDTIFG
        if (B1 == 1) { // Button up
            LED1 = 1; // LED1 off
        } else { // Button down
            WDTCTL = WDTPW | WDTCONFIG; // Feed/pet/kick/clear watchdog
            LED1 = 0; // LED1 on
        }
    }
}
The watchdog can be used as an interval timer if its protective function is not desired.

Set the WDTTMSEL bit in WDTCTL for interval timer mode. The periods are the same as before and again WDTIFG is set when the timer reaches its limit, but no reset occurs.

The counter rolls over and restarts from 0. An interrupt is requested if the WDTIE bit in the special function register IE1 is set. This interrupt is maskable and as usual takes effect only if GIE is also set.

The watchdog timer has its own interrupt vector, which is fairly high in priority but not at the top. It is not the same as the reset vector, which is taken if the counter times out in watchdog mode.

The WDTIFG flag is automatically cleared when the interrupt is serviced.
This is the most versatile, general-purpose timer in the MSP430 and is included in all devices.

There are two main parts to the hardware:

- **Timer block:** The core, based on the 16-bit register TAR. There is a choice of sources for the clock, whose frequency can be divided down (prescaled). The timer block has no output but a flag TAIFG is raised when the counter returns to 0.

- **Capture/compare channels:** In which most events occur, each of which is based on a register TACCRn. They all work in the same way with the important exception of TACCR0. Each channel can
  
  - **Capture an input,** which means recording the “time” (the value in TAR) at which the input changes in TACCRn; the input can be either external or internal from another peripheral or software.
  
  - **Compare** the current value of TAR with the value stored in TACCRn and update an output when they match; the output can again be either external or internal.
  
  - **Request an interrupt** by setting its flag TACCRn CCIFG on either of these events; this can be done even if no output signal is produced.
  
  - **Sample** an input at a compare event; this special feature is particularly useful if Timer_A is used for serial communication in a device that lacks a dedicated interface.
The simplest way of generating a fixed delay with Timer_A is to wait for TAR to overflow. We need only the Timer_A Control Register TACTL.

<table>
<thead>
<tr>
<th>15</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>w-(0)</td>
<td>rw-(0)</td>
</tr>
</tbody>
</table>

Figure 4.8: The Timer_A Control Register TACTL. Gray bits are unused.
Three items are given for each bit:

- Its position in the word, which should not be needed (use symbolic names instead).
- Its name, which is defined in the header file and should be known to the debugger; some bits are not used, which we show by a gray fill.
- The accessibility and initial condition of the bit; here they can all be read and written with the exception of TACLR, where the missing r indicates that there is no meaningful value to read. The (0) shows that each bit is cleared after a power-on reset (POR).
The user’s guide goes on to describe the function of each bit or group of bits:

- **Timer_A clock source select, TASSELx**: There are four options for the clock: the internal SMCLK or ACLK or two external sources. We use SMCLK because it is always available, which needs TASSELx = 10.

- **Input divider, IDx**: The frequency of the clock can be divided before it is applied to the timer, which extends the period of the counter. We use the maximum factor of eight, which needs IDx = 11.

- **Mode control, MCx**: The timer has four modes. By default it is off to save power. We first use the simplest Continuous mode, in which TAR simply counts up through its full range of 0x0000–0xFFFF and repeats. This needs MCx = 10.

- **Timer_A clear, TACLR**: Setting this bit clears the counter, the divider, and the direction of the count (it can go both up and down in up/down mode). The bit is automatically cleared by the timer after use. It is usually a good idea to clear the counter whenever the timer is reconfigured to ensure that the first period has the expected duration.

- **Timer_A interrupt enable, TAIE**: Setting this bit enables interrupts when TAIFG becomes set. We do not use this here.

- **Timer_A interrupt flag, TAIFG**: This bit can be modified by the timer itself or by a program. It is raised (set) by the timer when the counter becomes 0. In continuous mode this happens when the value in TAR rolls over from 0xFFFF to 0x0000. An interrupt is also requested if TAIE has been set. The program must clear TAIFG so that the next overflow can be distinguished.
- The sub-main clock SMCLK runs at the same speed as MCLK by default, which is 800 KHz for example.
- If this were used to clock the timer directly, the period would be $=2^{16}/800\text{KHz} \approx 0.08 \text{s}$.
- We want about 0.5 s and therefore divide the frequency of the clock by 8 using IDx. This gives a delay of about 0.64 s, close enough.
// timrled1.c - toggles LEDs with period of about 1.3s
// Poll free -running timer A with period of about 0.65s
// Timer clock is SMCLK divided by 8, continuous mode
// Olimex 1121STK, LED1,2 active low on P2.3,4
#include <io430x11x1.h> // Specific device
// Pins for LEDs
#define LED1 BIT3
#define LED2 BIT4
void main (void)
{
    WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
    P2OUT = ~LED1; // Preload LED1 on, LED2 off
    P2DIR = LED1|LED2; // Set pins for LED1,2 to output
    TACTL = MC_2|ID_3|TASSEL_2|TACLR; // Set up and start Timer A
    // Continuous up mode, divide clock by 8, clock from SMCLK, clear timer
    for (;;) { // Loop forever
        while (TACTL_bit.TAIFG == 0) { // Wait for overflow
            // doing nothing
            TACTL_bit.TAIFG = 0; // Clear overflow flag
            P2OUT ^= LED1|LED2; // Toggle LEDs
        } // Back around infinite loop
    }
}
More tasks could be added here, provided that they do not take longer than the period of the timer. The result is a *paced loop*, a straightforward structure for a program that carries out a sequence of tasks at regular intervals.

Nowadays it would be unusual to pace the loop by polling the timer; instead the MCU would save energy by entering a low-power mode after it had completed the tasks and wait for the timer to wake it again.
Timer_A in Up Mode

- Finer control over the delay is obtained by using the timer in Up mode rather than continuous mode. The maximum desired value of the count is programmed into another register, TACCR0. In this mode TAR starts from 0 and counts up to the value in TACCR0, after which it returns to 0 and sets TAIFG.
- Thus the period is TACCR0+1 counts
- Here the clock has been divided down to 100 KHz so we need 50,000 counts for a delay of 0.5 s and should therefore store 49,999 in TACCR0.

TACCR0 = 49999; // Upper limit of count for TAR
TACTL = MC_1|ID_3|TASSEL_2|TACLR;
// Set up and start Timer A
// "Up to CCR0" mode, divide clock by 8, clock from SMCLK, clear timer
A pretty application of the delay is a random light show on the LEDs. Of course this is rather limited with only two LEDs but the principle can be applied to bigger displays. This again uses a delay set by the timer but requires a calculation for the next pattern to display.

Figure 4.10: A shift register with feedback through an exclusive-OR gate from the last two stages, used to generate a pseudorandom stream of bits.
The circuit without the exclusive-OR gate and its connections is a plain *shift register*. A *D* flip-flop simply reads the value on its *D* input at a clock transition and transfers it to its *Q* output. Thus the value in flip-flop 0 is transferred to flip-flop 1 after a clock transition.

At the same time the value in flip-flop 1 is transferred to flip-flop 2 and so on. The pattern of bits simply shifts one place to the left in each clock cycle. An input is applied to the first flip-flop, 0.

A pseudorandom sequence requires more complicated feedback. The simplest method, shown in the figure, is to take the feedback from an exclusive-OR gate connected to the outputs of the last two stages.

The counter must therefore be “seeded” with a nonzero value. The counter in

Figure with *N* = 4 gives the sequence 0001, 0010, 0100, 1001, 0011, 0110, 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000 and repeat
Program to produce a pseudorandom bit sequence by simulating a shift register with feedback.

// random1.c - pseudorandom sequence on LEDs Poll timer A in Up mode with period of about 0.5s
// Timer clock is SMCLK divided by 8, up mode, period 50000 Olimex 1121STK, LED1,2 active low on P2.3,4
// -----------------------------------------------------------------------------
#include <io430x11x1.h> // Specific device
#include <stdint.h> // For uint16_t
#define LED1 BIT3 // Pins for LEDs
#define LED2 BIT4
// Parameters for shift register; length <= 15 (4 is good for testing)
#define REGLENGTH 15
#define LASTMASK ((uint16_t)(BIT0 << REGLENGTH))
#define NEXTMASK ((uint16_t)(BIT0 << (REGLENGTH -1)))

void main (void) {
    WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
    P2OUT = LED1|LED2; // Preload LEDs off
    P2DIR = LED1|LED2; // Set pins with LEDs to output
    TACCR0 = 49999; // Upper limit of count for TAR
    TACTL = MC_1|ID_3|TASSEL_2|TACLR; // Set up and start Timer A
    // "Up to CCR0" mode, divide clock by 8, clock from SMCLK, clear timer
    pattern = 1;
    for (;;) { // Loop forever
        while (TACTL_bit.TAIFG == 0) { // Wait for timer to overflow
            } // doing nothing
        TACTL_bit.TAIFG = 0; // Clear overflow flag
        P2OUT = pattern; // Update pattern (lower byte)
        pattern <<= 1; // Shift for next pattern
        // Mask two most significant bits, simulate XOR using switch, feedback
        switch (pattern & (LASTMASK|NEXTMASK)) {
            case LASTMASK:
            case NEXTMASK:
                pattern |= BIT0; // XOR gives 1
                break;
            default:
                pattern &= ~BIT0; // XOR gives 0
                break;
        }
    } // Back around infinite loop
}
- **ACLK**: Auxiliary clock. The signal is sourced from LFXT1CLK with a divider of 1, 2, 4, or 8. (The calibration program for the serial link sets the divider to 4, but after the calibration it can be changed to any other values.) ACLK can be used as the clock signal for Timer A and Timer B.

- **MCLK**: Master clock. The signal can be sourced from LFXT1CLK, XT2CLK (if available), or DCOCLK with a divider of 1, 2, 4, or 8. MCLK is used by the CPU and system.

- **SMCLK**: Sub-main clock. The signal is sourced from either XT2CLK (if available), or DCOCLK with a divider of 1, 2, 4, or 8. SMCLK can be used as the clock signal for Timer A and Timer B.
DCOCTL, DCO Control Register

<table>
<thead>
<tr>
<th>DCOx Bits</th>
<th>MODx Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>4-0</td>
</tr>
</tbody>
</table>

**DCOx Bits**
DCO frequency select. These bits select which of the eight discrete DCO frequencies of the RSELx setting is selected.

**MODx Bits**
Modulator selection. These bits define how often the $f_{DCO+1}$ frequency is used within a period of 32 DCOCLK cycles. During the remaining clock cycles ($32-\text{MOD}$) the $f_{DCO}$ frequency is used. Not useable when DCOx=7.
# BCSTL1 Basic Clock System Control

Register 1

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5 (x)</th>
<th>Bit 4 (x)</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XT2OFF</td>
<td>XTS</td>
<td>DIVAx</td>
<td>XT5V</td>
<td>RSELx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rw-(1)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**XT2OFF** Bit 7
- XT2 off. This bit turns off the XT2 oscillator
- 0: XT2 is on
- 1: XT2 is off if it is not used for MCLK or SMCLK.

**XTS** Bit 6
- LFXT1 mode select.
- 0: Low frequency mode
- 1: High frequency mode

**DIVAx** Bits 5-4
- Divider for ACLK
- 00: /1
- 01: /2
- 10: /4
- 11: /8

**XT5V** Bit 3
- Unused. XT5V should always be reset.

**RSELx** Bits 2-0
- Resistor Select. The internal resistor is selected in eight different steps.
- The value of the resistor defines the nominal frequency. The lowest nominal frequency is selected by setting RSELx=0.
## BCSTL2 Basic Clock System Control

### Register 2

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>SELM&lt;sub&gt;x&lt;/sub&gt;</th>
<th>DIVM&lt;sub&gt;x&lt;/sub&gt;</th>
<th>SELS</th>
<th>DIVS&lt;sub&gt;x&lt;/sub&gt;</th>
<th>DCOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-0</td>
<td>rw-0</td>
</tr>
<tr>
<td>5-4</td>
<td>SELM&lt;sub&gt;x&lt;/sub&gt;</td>
<td>DIVM&lt;sub&gt;x&lt;/sub&gt;</td>
<td>SELS</td>
<td>DIVS&lt;sub&gt;x&lt;/sub&gt;</td>
<td>DCOR</td>
</tr>
<tr>
<td>3-2</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
</tr>
<tr>
<td>1-0</td>
<td>rw-(0)</td>
<td>rw-(0)</td>
<td>rw-0</td>
<td>rw-0</td>
<td>rw-0</td>
</tr>
</tbody>
</table>

#### SELM<sub>x</sub>
- **Bits**: Select MCLK. These bits select the MCLK source.
- **7-6**: 00 DCOCLK, 01 DCOCLK, 10 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK when XT2 oscillator not present on-chip, 11 LFXT1CLK

#### DIVM<sub>x</sub>
- **BitS**: Divider for MCLK
- **5-4**: 00 /1, 01 /2, 10 /4, 11 /8

#### SELS
- **Bit 3**: Select SMCLK. This bit selects the SMCLK source.
- **0**: DCOCLK
- **1**: XT2CLK when XT2 oscillator present on-chip. LFXT1CLK when XT2 oscillator not present on-chip.

#### DIVS<sub>x</sub>
- **BitS**: Divider for SMCLK
- **2-1**: 00 /1, 01 /2, 10 /4, 11 /8

#### DCOR
- **Bit 0**: DCO resistor select
- **0**: Internal resistor
- **1**: External resistor
Configuring the Timer

- **TACTL** – Timer A Control Register
  This register used to configure how the timer runs
- **TACCTL0** – Capture/Compare Control Register
  For enabling and disabling TimerA0 interrupt
- **TACCR0** – Capture/Compare Register
  This registers holds the value YOU define to configure the timing
Modes of Operation: Up Mode

- Timer counts UP from zero to TACCRO
- Interrupt occurs when timer goes back to zero
- Useful for periods other than 0xFFFF
Modes of Operation: Up Mode

- Timer Clock
- CCR0
- CCR0
- 0h
- 1h
- 0h
- Set TAIFG
- Set TACCR0 CCIFG
Continuous Mode

- Timer counts from 0 to 0xFFFF
- Fewer timing errors because timer never stops – keeps counting up until it reaches 0xFFFF and rolls over to 0 and keeps going.
- The ACTUAL VALUE of the timer does not matter – only the RELOAD VALUE matters – this controls the period of the interrupt.
- Interrupt DOES NOT OCCUR AT 0 OR 0xFFFF!
- Occurs when timer reaches current TACCR0 value!
Modes of Operation: Continuous Mode

Timer Clock

Timer

Set TAIFG
Continuous Mode

Continuous Timer Mode 2:
Say that reload value is 0x6000

1) Timer reaches 0x6000, fires interrupt, inside ISR we ADD the period to the CURRENT VALUE of the timer:
   TACCR0 += 0x6000
   Therefore TACCR0 = 0xC000, and next interrupt will fire when timer reaches this value.

2) Timer now reaches 0xC000, fires interrupt, inside interrupt
   TACCR0 += 0x6000
   New TACCR0 = 0x2000 (rolls over 0xFFFF)

3) Timer rolls over 0xFF and now reaches 0x2000. Cycle repeats.
Modes of operation: Up Down mode

- Timer counts from 0 to TACCRO, then back down to 0
- Used when timer period must be different from 0xFFFF and when pulse needs to be symmetric
- Good for driving motors (ON pulse to control speed)
Modes of operation: Up Down mode

Graph showing the modes of operation for the TACCR0 register with a timeline illustrating the interaction between Timer Clock, Timer, Up/Down, Set TAI FG, and Set TACCR0 CCFG.
Timer_A Registers

- **TACTL, Timer_A Control Register (PART 1)**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Unused**

- **Bits**: Unused
- **15-10**

**TASSELx**

- **Bits**: Timer_A clock source select
- **9-8**
  - 00 TACLK
  - 01 ACLK
  - 10 SMCLK
  - 11 INCLK
## TACTL, Timer_A Control Register

### (PART 2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td><strong>IDx</strong> Bits</td>
</tr>
<tr>
<td>5-4</td>
<td><strong>MCx</strong> Bits</td>
</tr>
<tr>
<td>3</td>
<td><strong>Unused</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>TACLR</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>TAIE</strong></td>
</tr>
<tr>
<td>0</td>
<td><strong>TAIFG</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>00</td>
<td>/1</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>/2</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>/4</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>/8</td>
</tr>
<tr>
<td>5-4</td>
<td>00</td>
<td>Stop mode: the timer is halted</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>Up mode: the timer counts up to TACCR0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Continuous mode: the timer counts up to 0FFFFh</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Up/down mode: the timer counts up to TACCR0 then down to 0000h</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Unused</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Interrupt disabled</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Interrupt enabled</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>No interrupt pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interrupt pending</td>
</tr>
</tbody>
</table>
# TACCTLx, Capture/Compare Control Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CMx</td>
<td>Capture mode</td>
</tr>
<tr>
<td>14</td>
<td>CCISx</td>
<td>Capture mode</td>
</tr>
<tr>
<td>13</td>
<td>SCS</td>
<td>Capture mode</td>
</tr>
<tr>
<td>12</td>
<td>SCCI</td>
<td>Capture mode</td>
</tr>
<tr>
<td>11</td>
<td>Unused</td>
<td>Capture mode</td>
</tr>
<tr>
<td>10</td>
<td>CAP</td>
<td>Capture mode</td>
</tr>
<tr>
<td>9</td>
<td>OUTMODx</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of</td>
</tr>
<tr>
<td>8</td>
<td>CCIE</td>
<td>the corresponding CCF flag.</td>
</tr>
<tr>
<td>7</td>
<td>CCIFG</td>
<td>Capture/compare interrupt flag</td>
</tr>
<tr>
<td>6</td>
<td>OUT</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of</td>
</tr>
<tr>
<td>5</td>
<td>COV</td>
<td>the corresponding CCF flag.</td>
</tr>
<tr>
<td>4</td>
<td>CCI</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of</td>
</tr>
<tr>
<td>3</td>
<td>r</td>
<td>the corresponding CCF flag.</td>
</tr>
<tr>
<td>2</td>
<td>rw-0</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of</td>
</tr>
<tr>
<td>1</td>
<td>rw-0</td>
<td>the corresponding CCF flag.</td>
</tr>
<tr>
<td>0</td>
<td>rw-0</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of</td>
</tr>
</tbody>
</table>

**CAP**
- Bit 8: Capture mode
  - 0: Compare mode
  - 1: Capture mode

**CCIE**
- Bit 4: Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.
  - 0: Interrupt disabled
  - 1: Interrupt enabled

**CCIFG**
- Bit 0: Capture/compare interrupt flag
  - 0: No interrupt pending
  - 1: Interrupt pending
Example 1

Continuous Mode
Output pin P6.0 with toggle rate = 32768/(2*50) = 328Hz

```c
#include "include/include.h"
#include "include/hardware.h"
void main ( void )
{
    WDTCTL = WDTPW + WDTHOLD; // Stop WDT
    P6DIR |= 0x01; // P6.0 output
    CCTL0 = CCIE; // CCR0 interrupt enabled
    CCR0 = 50;
    TACTL = TASSEL_1 + MC_2; // ACLK, contmode
    eint(); // Enable the global interrupt
    //or _BIS_SR(LPM0_bits + GIE);
    LPM0; // Enter low power mode or wait in a loop
}
// Timer_A TACCR0 interrupt vector handler
interrupt (TIMERA0_VECTOR) TimerA_procedure(void){
    P6OUT ^= 0x01; // Toggle P6.0
    CCR0 += 50; // Add offset to CCR0
}
```
Example 2

- **Up Mode**
- **Output pin P6.0 with toggle rate = \( \frac{32768}{(2 \times 50)} \) = 328Hz**

```c
#include "include/include.h"
#include "include/hardware.h"
void main ( void )
{
    WDTCTL = WDTPW + WDTHOLD; // Stop WDT
    P6DIR |= 0x01; // P6.0 output
    CCTL0 = CCIE; // CCR0 interrupt enabled
    CCR0 = 50-1;
    TACTL = TASSEL_1 + MC_1; // ACLK, upmode
    _BIS_SR(LPM0_bits + GIE); // Enable the global interrupt and enter LPM0
}
// Timer_A TACCR0 interrupt vector handler
interrupt (TIMERA0_VECTOR) TimerA_procedure ( void ){
P6OUT ^= 0x01; // Toggle P6.0
}
```