Week 5
FUNDAMENTALS OF INTERFACING AND TIMERS for MSP430

Hacettepe University
Elements in Basic MCU Interface

- Power Source
  - Feeds CPU and peripherals
- Clock Oscillators
  - System synchronization
- Power-on Reset
  - Physical reset hardware
- Booting Function
  - System configuration and initialization

Fig. 6.1 Components in a basic CPU interface: power source, clock generator, power-on-reset, and boot sequence
Main Function

- Provide Power to CPU and Surrounding Electronics
- Establish reference levels for internal device operation

Basic Requirements

- Steady voltage source
- Sufficient current capability
- Load regulation
- Power quality

Implications

- System functionality and integrity
- Signal compatibility

Fig. 6.2 Structure of a typical MCU power supply
Absolute Maximum Ratings
- Levels of stress that if exceeded will cause permanent damage to the device
- If used for extended periods will affect device reliability
- DO NOT design for operating devices at these levels

Recommended Operating Conditions
- Manufacturer’s recommended levels for reliable operation
- Conditions the application circuit should provide to device for it to function as intended
- Shall be used for design calculations
Sample Abs. Max. Ratings Specs

- Voltage Levels
- Maximum and minimum supply values
- Applied to any pin
- Diode Current
- ESD clamping diode current
- Storage Temperature

Table 6.1: Absolute maximum ratings for MSP430G2231

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Condition</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage applied at ( V_{cc} ) to ( V_{ss} )</td>
<td></td>
<td>(-0.3) to 4.1V</td>
</tr>
<tr>
<td>Voltage applied to any pin</td>
<td></td>
<td>0.3V to ( V_{cc} + 0.3V )</td>
</tr>
<tr>
<td>Diode current at any device pin</td>
<td></td>
<td>( \pm2mA )</td>
</tr>
<tr>
<td>Storage temperature range, ( T_{stg} )</td>
<td>Unprogrammed device</td>
<td>(-55^\circ)C to 150(^\circ)C</td>
</tr>
<tr>
<td></td>
<td>Programmed device</td>
<td>(-40) to 85(^\circ)C</td>
</tr>
</tbody>
</table>
• Recommendation
  • Stay below absolute maximum ratings
  • Ground level specified independent of the Vcc

<table>
<thead>
<tr>
<th>Symb.</th>
<th>Parameter</th>
<th>Condition</th>
<th>Min.</th>
<th>Max.</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vcc</td>
<td>Supply voltage</td>
<td>Program execution</td>
<td>1.8</td>
<td>3.6</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flash programming</td>
<td>2.2</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>Vss</td>
<td>Supply voltage</td>
<td></td>
<td>0</td>
<td>0</td>
<td>V</td>
</tr>
<tr>
<td>TA</td>
<td>Operating temp.</td>
<td>Free air</td>
<td>~40</td>
<td>85</td>
<td>°C</td>
</tr>
<tr>
<td>f_clk</td>
<td>Clock frequency</td>
<td>Vcc = 1.8V, Duty cycle = 50% ± 10%</td>
<td>dc</td>
<td>4.15</td>
<td>MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vcc = 2.7V, Duty cycle = 50% ± 10%</td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vcc = 3.3V, Duty cycle = 50% ± 10%</td>
<td></td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Minimum Power Dissipation
- Obtained at minimum $V_{DD}$
- Power varies with $V_{DD}^2$

Maximum $f_{clk}$ Limit
- $f_{clk\ max}$ is usually limited by the chosen $V_{DD}$ level
- Direct relationship

Choose Wisely
- Use the minimum frequency and voltage necessary for proper functionality
Power Supply Capacity

- Regulator Capacity ($I_R$)
  \[ I_R \geq I_L = \sum_{i=1}^{n} I_{Li} \]  
  (1)
  where $I_{Li} = \text{current from load } i = 1,2,\ldots,n$

- Input voltage ($V_{NR}$)
  \[ V_{NR} \geq (V_R + V_D) \]  
  (2)
  where
  - $V_R = \text{regulated voltage and}$
  - $V_D = \text{dropout voltage}$

- Non-regulated capacity
  \[ I_{NR} \geq (I_R + I_{gnd}) \]  
  (3)

- Non-regulated power:
  \[ P_{NR} = V_R (I_R + I_{gnd}) \]  
  (4)

- Power to the load:
  \[ P_R = V_R \cdot I_L \]  
  (5)

- Power Efficiency:
  \[ Eff = \left( \frac{P_R}{P_{NR}} \right) \times 100\% \]  
  (6)
Example 6.1: Consider using a standard 9V alkaline battery for feeding a 3.3V, 120mA load via a linear voltage regulator uA78M33C. Estimate the approximated battery life and usage efficiency assuming a constant load current.

Regulator data:
\[ V_o = 3.3V \quad V_D = 2.0V \quad I_{out} = 500mA \]
\[ I_{bias} = I_{gnd} = 6mA \]

Solution: Total load seen by battery (I_{NR})
\[ I_{NR} = I_L + I_{gnd} = 126mA \leq I_{out} \]

From battery datasheet: \[ Q_{bat} = 480mAh \] at a 100mA discharge rate. At 300mA, \[ Q_{bat} \] drops to 380mAh. Interpolating we get \[ Q_{bat}
\]

\[ t_{bat} = \frac{Q_{bat}}{I_{NR}} = \frac{447mAh}{126mA} = 3.55\ hours \]

\[ Eff = \frac{V_R \times I_R}{V_{NR} \times (I_R + I_{gnd})} \times 100\% = \frac{3.3V \times 120mA}{9V \times (120 + 6)mA} = 34.92\% \]
Coping with Power Supply Noise
- Reducing the effect of noise in the power distribution lines

Common Methods
- Bypassing Techniques
- Source Decoupling
- Wise Power Distribution

Combine ALL of them
- Plan the power distribution network
- Power lines noise increases with:
  - Clock frequency
  - Power distribution loop length
  - Dirty power supplies
Bypassing refers to the act of reducing a high frequency current flow in a circuit path by adding a shunting component that reacts to the target frequency. The most commonly used shunting devices in microprocessor-based designs are bypassing capacitors.

A bypass capacitor reduces the rate of change of the current circulating in the power line by providing a high-frequency, low impedance path to the varying load current. Two factors determine the effectiveness of a bypassing capacitor: size and location.
Clock Issues

- **Clock Frequency**
- **Clock Duty Cycle**

The clock duty cycle defines the ratio of thigh to the period of the clock signal

- **Clock Stability**

Example 6.2 Consider a 12MHz clock signal that exhibits a ±10% deviation from its nominal value. Two applications employing such a clock signal are analyzed: a dynamic display that sets a 60Hz refresh ratio from this clock and a real-time clock slated to run uninterruptedly for weeks. Evaluate the impact of the clock accuracy on each system.

**Solution:** The impact of the clock deviation will be analyzed independently for each system.

**Impact on the display system:** A 10% frequency deviation would translate into an equally proportional deviation in the refresh rate, implying that the actual rate could be 6Hz off the target value. In the worst case, assuming a 10% frequency loss, the refresh ratio would be 54Hz. Considering that for persistence of vision, the human eye only requires 24Hz of refresh ratio to perceive motion, this 10% change of frequency can be deemed as negligible.

**Impact on the RTC:** A 10% deviation in frequency would cause the RTC to drift from the actual time at a rate of 6s each minute or 2h and 24min per day. At the end of only one week, assuming a negative δ fCLK, the error would accumulate to 16.8h, which would be totally unacceptable.
• **Clock Jitter** Clock Jitter refers to the uncertainty in the periodicity of a clock signal.

**Example 6.3** Consider a microprocessor with a total system clock jitter specified at 150ps under the JESD65B standard. This specification establishes that the total time deviation in the signal period resulting from the sum of all deterministic and random sources in the clock frequency over a minimum of $10^4$ cycles cannot exceed 150ps. For some devices, the number of cycles specified in the JESD65B standard establishes a bare minimum. A commonly used number is $10^5$ cycles, but for some devices it can reach as much as $10^{12}$ cycles.

**Clock Drift** Frequency drift refers to the linear component of a systematic change in the frequency of an oscillator over time. For example, a 4MHz oscillator with an age induced drift of 20PPM per year will deviate its frequency by 80Hz every year.
Choosing Clock

- What is the fastest event the system will need to handle?
- Has the value of VDD been assigned?
- What peripherals will share the same clock frequency?
- How precise does the clock need to be?
- What are the capabilities of the Clock System in my MCU?

<table>
<thead>
<tr>
<th>Freq. (MHz)</th>
<th>Typical application</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.032768</td>
<td>Real-time clocks. Allows binary division to 1.0000Hz ($2^{15} \times 1$ Hz)</td>
</tr>
<tr>
<td>1.843200</td>
<td>UART clock. (16 × 115, 200 baud or 96 × 16 × 1, 200 baud)</td>
</tr>
<tr>
<td>2.457600</td>
<td>UART clock. (64 × 38, 400 baud or 2, 048 × 1, 200 baud)</td>
</tr>
<tr>
<td>3.276800</td>
<td>Allows binary division to 100 Hz (32, 768 × 100 Hz, or $2^{15} \times 100$ Hz)</td>
</tr>
<tr>
<td>3.579545</td>
<td>NTSC M color subcarrier and DTMF generators</td>
</tr>
<tr>
<td>3.686400</td>
<td>UART clock (2 × 1.8432 MHz)</td>
</tr>
<tr>
<td>4.096000</td>
<td>Allows binary division to 1kHz ($2^{12} \times 1$ kHz)</td>
</tr>
<tr>
<td>4.194304</td>
<td>Real-time clocks, divides to 1 Hz signal ($2^{22} \times 1$ Hz)</td>
</tr>
<tr>
<td>6.144000</td>
<td>UART baud rates up to 38,400.</td>
</tr>
<tr>
<td>6.553600</td>
<td>Allows binary division to 100 Hz ($2^{16} \times 100$ Hz)</td>
</tr>
<tr>
<td>7.372800</td>
<td>UART clock (4 × 1.8432 MHz)</td>
</tr>
<tr>
<td>9.216000</td>
<td>Allows integer division to 1,024 kHz ($2^{10}$)</td>
</tr>
<tr>
<td>11.059200</td>
<td>UART clock (6 × 1.8432 MHz)</td>
</tr>
</tbody>
</table>
The earliest generation of the MSP430 device family, the MSP430x3xx uses a *Frequency-Locked Loop* clock module as system clock generator. This clocking system consists of two oscillators:

- a Pierce crystal oscillator and a frequency stabilized & RC-based, Digitally Controlled Oscillator (DCO). The DCO is locked to a multiple of the crystal frequency, forming a frequency-locked loop (FLL). frequency stability and quick startup.

This clocking system fundamentally provides two clock signals: the *Main Clock* (MCLK), taken from the DCO output and an *Auxiliary Clock* signal (ACLK), taken out from the crystal oscillator.

A buffered, software selectable output XBUF is also available, that provides as output either MCLK, ACLK, ACLK/2, or ACLK/4. The XBUF clock can also be turned off via software. The MCLK signal is used to drive the CPU, while ACLK and XBUF can be software selected to drive peripherals.
The second generation of MSP430, the MSP430x4xx, features an improved clock generator designated the **FLL+ Clock Module**.

LFXT1, DCO and now an XT2!

The FLL+ module can provide four clock signals to the system: ACLK, ACLK/n (n = 1, 2, 4, or 8), MCLK, and SMCLK.

Fig. 6.17 Simplified block diagram of FLL+ clock module in MSP430x4xx devices
Generations x1xx and x2xx featured a modified clock module with respect to the design included in earlier generations. This new clock generator is designated as \textit{Basic Clock Module}.

DCO is now redesigned to operate in open loop (no FLL), while offering a better frequency stability without the need of an external crystal.

In addition, only three clock signals are made available: ACLK, MCLK, and SMCLK.

\textbf{Fig. 6.18} Simplified block diagram of basic clock module in MSP430x1xx devices
# Summary of Clocks in MSP

## Table 6.4: Summary of clock configurations for MSP430 generations x1xx through x6xx

<table>
<thead>
<tr>
<th>MSP430</th>
<th>Name</th>
<th>Oscillators</th>
<th>Clock signals</th>
<th>Max freq. (MHz)</th>
<th>DCO type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3xx</td>
<td>FLL</td>
<td>XTAL, DCO</td>
<td>MCLK, ACLK, XBUF</td>
<td>0.032</td>
<td>FLL</td>
</tr>
<tr>
<td>x4xx</td>
<td>FLL+</td>
<td>LFXT1, DCO, XT2</td>
<td>MCLK, SMCLK, ACLK, ACLK/n</td>
<td>8.000</td>
<td>FLL</td>
</tr>
<tr>
<td>x1xx</td>
<td>BCM</td>
<td>LFXT1, DCO, XT2</td>
<td>MCLK, ACLK, SMCLK</td>
<td>8.000</td>
<td>DCO</td>
</tr>
<tr>
<td>x2xx</td>
<td>BCM+</td>
<td>LFXT1, DCO, XT2, VLO</td>
<td>MCLK, ACLK, SMCLK</td>
<td>16.000</td>
<td>DCO</td>
</tr>
<tr>
<td>x5xx</td>
<td>UCS</td>
<td>LFXT1, DCO, XT2, REFO, VLO, MODOSC</td>
<td>MCLK, ACLK, SMCLK</td>
<td>32.000</td>
<td>FLL/DCO</td>
</tr>
<tr>
<td>x6xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright 2009 Texas Instruments
All Rights Reserved
Power-On Reset

- Loading the program counter register (PC) with the address of the first instruction to be executed. This causes execution of the first instruction in the booting sequence.
- Disabling the reception of maskable interrupts by the CPU.
- Clearing the status register (SR). The specific value loaded into the SR changes from one processor or MCU to another.
- Initializing some or all system peripherals (list changes for specific devices). For example, many MCUs set all their I/O pins to input mode, timers are initialized to zero, and the default CPU operating mode is selected.
- Canceling any bus transaction in progress and returning control to the default bus master (details of bus Arbitration in Chap. 7).

The RESET signal in an embedded system is generated through a specialized circuit called a **power-on reset circuit** or POR for short.

![Typical CPU RESET timing](image)

*Fig. 6.24 Typical CPU RESET timing*
Power-On Reset

\[ t_{\text{reset}} = RC \cdot \ln \left( \frac{V_{DD}}{V_{DD} - V_T^+} \right) \]

Fig. 6.25 A POR circuit based on a one-shot topology

Copyright 2009 Texas Instruments
All Rights Reserved
After a POR, the initial conditions in an MSP430 device are the following:
- The functionality of the RST/NMI pin is set to RST.
- GPIO pins in all ports are configured as inputs.
- The processor status register (SR) is loaded with the reset value, which clears the V, N, Z, and C flags, disables all maskable interrupts (GIE = 0), and the CPU is set to active mode, cancelling any low-power mode previously set.
- The watchdog timer is activated in watchdog mode.
- The program counter (PC) is loaded with the address pointed by the reset vector (0FFFEh). Setting the reset vector contents to 0FFFFh, disables the device, entering a low power mode.
- Particular peripheral modules and registers are also affected, depending on the specific device being used.
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 timer:** 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.

<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>WDT-HOLD</td>
<td>WDT-NMIES</td>
<td>WDTNMI</td>
<td>WDT-TMSEL</td>
<td>WDT-CNTCL</td>
<td>WDTSSSEL</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>rw-(0)</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 WDT HOLD 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
    // WDTSESEL 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
        }
    }
}
Watchdog as an Interval Timer

- 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.
Timer_A

- 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.

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 (;;) {

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 , feed back
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
MSP430 Basic Clock Module

- **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>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCOx</td>
<td>DCO frequency select. These bits select which of the eight discrete DCO frequencies of the RSELx setting is selected.</td>
</tr>
<tr>
<td>MODx</td>
<td>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-MOD) the $f_{DCO}$ frequency is used. Not useable when DCOx=7.</td>
</tr>
</tbody>
</table>
## BCSTL1 Basic Clock System Control
### Register 1

<table>
<thead>
<tr>
<th></th>
<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>XT2OFF</td>
<td>rw-(1)</td>
<td></td>
<td></td>
<td></td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XTS</td>
<td>rw-(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVAx</td>
<td>rw-(0)</td>
<td></td>
<td></td>
<td></td>
<td>rw-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XT5V</td>
<td></td>
<td>rw-0</td>
<td></td>
<td></td>
<td>rw-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSELx</td>
<td></td>
<td>rw-0</td>
<td></td>
<td></td>
<td>rw-0</td>
<td></td>
<td></td>
<td>rw-0</td>
</tr>
</tbody>
</table>

### Description:

**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 7-6</th>
<th>Bit 5-4</th>
<th>Bit 3</th>
<th>Bit 2-1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELMx</strong></td>
<td>Select MCLK. These bits select the MCLK source.</td>
<td>00 DCOCCLK 01 DCOCCLK</td>
<td>10 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK when XT2 oscillator not present on-chip.</td>
<td>11 LFXT1CLK</td>
</tr>
<tr>
<td><strong>DIVMx</strong></td>
<td>Divider for MCLK</td>
<td>00 /1 01 /2</td>
<td>10 /4</td>
<td>11 /8</td>
</tr>
<tr>
<td><strong>SELS</strong></td>
<td>Select SMCLK. This bit selects the SMCLK source.</td>
<td>0 DCOCCLK</td>
<td>1 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK when XT2 oscillator not present on-chip.</td>
<td></td>
</tr>
<tr>
<td><strong>DIVSx</strong></td>
<td>Divider for SMCLK</td>
<td>00 /1 01 /2</td>
<td>10 /4</td>
<td>11 /8</td>
</tr>
<tr>
<td><strong>DCOR</strong></td>
<td>DCO resistor select</td>
<td>0 Internal resistor</td>
<td>1 External resistor</td>
<td></td>
</tr>
</tbody>
</table>

**Bits**
- **SELMx**: 00 DCOCCLK, 01 DCOCCLK, 10 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK when XT2 oscillator not present on-chip, 11 LFXT1CLK
- **DIVMx**: 00 /1, 01 /2, 10 /4, 11 /8
- **SELS**: 0 DCOCCLK, 1 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK when XT2 oscillator not present on-chip
- **DIVSx**: 00 /1, 01 /2, 10 /4, 11 /8
- **DCOR**: 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

![Diagram showing modes of operation in up mode.](image)
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

![Modes of Operation Diagram](image-url)
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
Timer_A Interrupt Vectors

- CCR0 → CCIFG0 → CCIE0 → TACCR0 Interrupt Vector
- CCR1 → CCIFG1 → CCIE1
- CCR2 → CCIFG2 → CCIE2 → TACCR0 Interrupt Vector
- TAR Overflow
  → TAIFG → TAIE → TA/V
## Timer_A Registers

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

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Unused Bits
- Bits: 15-10
- Unused

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

## (PART 2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td><strong>IDX</strong> Components: These bits select the divider for the input clock.</td>
</tr>
<tr>
<td></td>
<td>00: /1</td>
</tr>
<tr>
<td></td>
<td>01: /2</td>
</tr>
<tr>
<td></td>
<td>10: /4</td>
</tr>
<tr>
<td></td>
<td>11: /8</td>
</tr>
<tr>
<td>5-4</td>
<td><strong>MCX</strong> Bit Components: Mode control. Setting MCX = 00h when Timer_A is not in use conserves power.</td>
</tr>
<tr>
<td></td>
<td>00: Stop mode: the timer is halted</td>
</tr>
<tr>
<td></td>
<td>01: Up mode: the timer counts up to TACCR0</td>
</tr>
<tr>
<td></td>
<td>10: Continuous mode: the timer counts up to 0FFFFh</td>
</tr>
<tr>
<td></td>
<td>11: Up/down mode: the timer counts up to TACCR0 then down to 0000h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><strong>TACLR</strong> Timer_A clear. Setting this bit resets TAR, the TACLK divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.</td>
</tr>
<tr>
<td>2</td>
<td><strong>TAIE</strong> Timer_A interrupt enable. This bit enables the TAIFG interrupt request.</td>
</tr>
<tr>
<td></td>
<td>0: Interrupt disabled</td>
</tr>
<tr>
<td></td>
<td>1: Interrupt enabled</td>
</tr>
<tr>
<td>0</td>
<td><strong>TAIFG</strong> Timer_A interrupt flag</td>
</tr>
<tr>
<td></td>
<td>0: No interrupt pending</td>
</tr>
<tr>
<td></td>
<td>1: Interrupt pending</td>
</tr>
</tbody>
</table>

Copyright 2009 Texas Instruments
All Rights Reserved
## 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>Compare 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 mode</td>
</tr>
<tr>
<td>8</td>
<td>CCIE</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.</td>
</tr>
<tr>
<td>4</td>
<td>CCI</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.</td>
</tr>
<tr>
<td>3</td>
<td>OUT</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.</td>
</tr>
<tr>
<td>2</td>
<td>COV</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.</td>
</tr>
<tr>
<td>0</td>
<td>CCIFG</td>
<td>Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.</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 (TIMER A0_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 (TIMER0_VECTOR) TimerA_procedure ( void ){
    P6OUT ^= 0x01; // Toggle P6.0
}
```