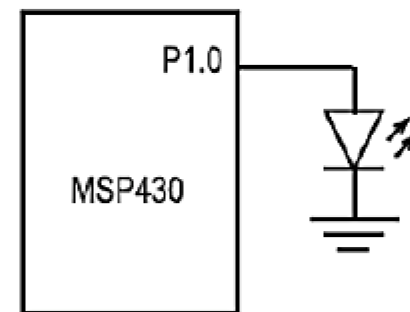
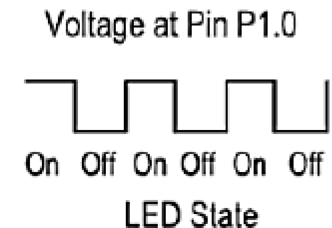
Example

Ex: Write an assembly language program to blink the LED connected to pin0 of PORT 1 (P1.0) of MSP430F5529 microcontroller. Use a decent amount of delay between ON and OFF states of the LED.

```
    mov.b #0x01,P1DIR ;P1.0 is output
here:
    xor.b #0x01,P1OUT ;Always complement P1.0
    mov.w #50000,R5 ;Load a big number for counting down
for1: ;loop to obtain delay between ON and OFF
    dec.w R5
    cmp #0,R5 ;check if R5 is 0
    jnz for1 ;continue till R5 is 0
end:
    jmp here
```





Subroutines

- Subroutines are similar to functions that can be called from any point of the program anytime.
- “`call` #Label” is the instruction that jumps the flow of the program to the desired point
- “Label” is the title of the subroutine. It can be anything in English characters.
- “`ret`” instruction returns the flow of the program where the subroutine is called. Apart from C/C++, It CANNOT return any value.
- Even though the subroutines are similar to the jump (conditional or unconditional) instructions, they are different since they return. There are no return mechanisms for jump instructions.
- Just like jump instructions, program flow may be through a subroutine even though it is not invoked.

Subroutines

Ex. Follow the flow for the program given below

```
        call #Subt ;Call Subt
        call #Sum ;Call Sum
        jmp Done ;jump to end, otherwise it continues
Sum:    mov.b #0x12,R7
        mov.b #0x34,R8
        add R7, R8
        ret ;Return to where it is invoked from
Subt:   mov.w #0x1234,R5
        mov.w #0x5678,R6
        sub R5, R6
        ret ;Return to where it is invoked from
Done: ;end of the program
```

Subroutines

Subroutines can really be useful for making the loops...

Ex:

```
        call   #Subt ;Call Subt
        call   #KST ;Kill Some Time
        call   #Sum ;Call Sum
        jmp Done ;jump to the end, otherwise it continues
Sum:    mov.b #0x12, R7
        mov.b #0x34, R8
        add R7, R8
        ret
KST:    mov.b #0x12, R9 ;Kill Some Time between the operations
Check:  dec r9
        jnz Check ;continue as long as Z is not 0
        ret
Subt:   mov.w #0x1234, R5
        mov.w #0x5678, R6
        sub R5, R6
        ret
Done: ;end of the program
```

Subroutines

Subroutines can also be nested…

Ex.

```
        call #Sum ;call Sum
        jmp Done ;jump to end, otherwise it continues
   Sum: mov.b #0x12,R7
        mov.b #0x34,R8
        add R7, R8
        call #KST ;Kill Some Time
        call #Subt ;call Subt
        ret
KST:    mov.b #0x12,R9 ;Kill Some Time between the operations
Check:  dec r9
        jnz Check ;continue as long as Z is not 0
        ret
Subt:   mov.w #0x1234,R5
        mov.w #0x5678,R6
        sub R5, R6
        ret
Done:  ;end of the program
```

EEE 204

Fundamentals of Interfacing

Asst. Prof. Dr Mahmut AYKAÇ

Fundamentals of Interfacing

Any microprocessor, needs a basic interface to become usable. This interface includes

Four fundamental elements:

- A power source to feed power to the CPU and system peripherals
- A clock generator to synchronize the system operation
- A power-on reset circuit (POR), to take the system to its initial state
- A booting function, to initiate the system software execution

Fundamentals of Interfacing

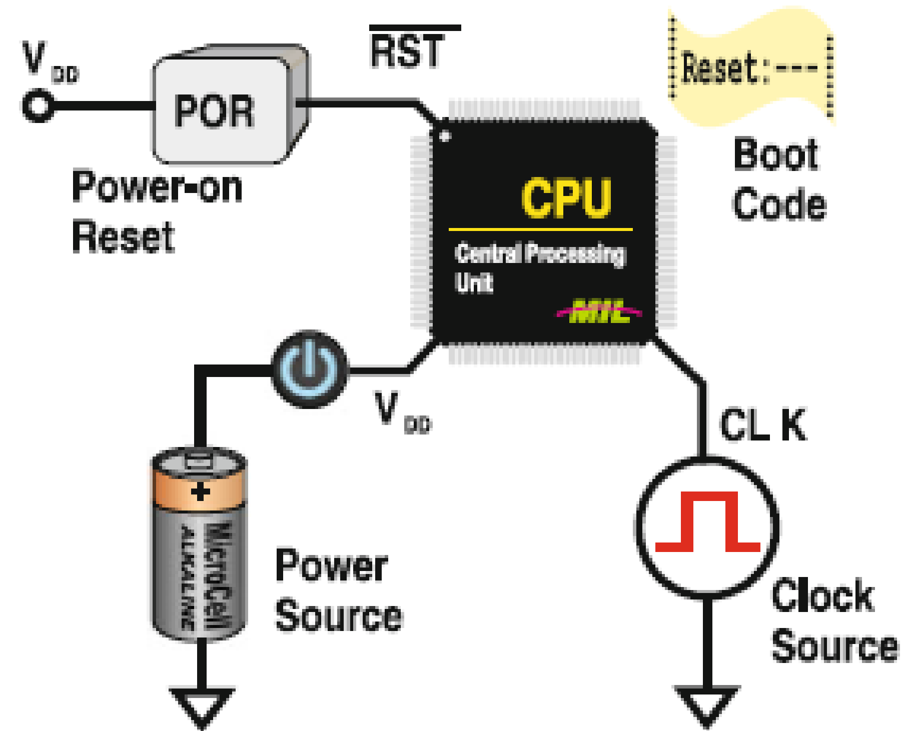


Figure. Components in a basic CPU interface

Processors' Power Sources

Regardless of the processor type, power sources need to be externally provided. They can be implemented from different sources: batteries, wall connected AC power outlet, or some form of energy collecting or scavenging device. In every case, selection and design criteria must be applied to properly feed power to the CPU and its peripherals.

Symb.	Parameter	Condition	Min.	Max.	Unit
V_{CC}	Supply voltage	Program execution	1.8	3.6	V
		Flash programming	2.2	3.6	
V_{SS}	Supply voltage		0	0	V
T_A	Operating temp.	Free air	−40	85	°C
f_{clk}	Clock frequency	$V_{CC} = 1.8$ V, Duty cycle $= 50\% \pm 10\%$	dc	4.15	MHz
		$V_{CC} = 2.7$ V, Duty cycle $= 50\% \pm 10\%$	dc	12	
		$V_{CC} = 3.3$ V, Duty cycle $= 50\% \pm 10\%$	dc	16	

Figure. Recommended operating conditions for MSP430G2231 as an example

Processors' Power Sources

A typical MCU power supply includes an unregulated power source and a voltage regulator to feed the load. If more than one voltage level were required, multiple regulators would be used, one for each different voltage level.

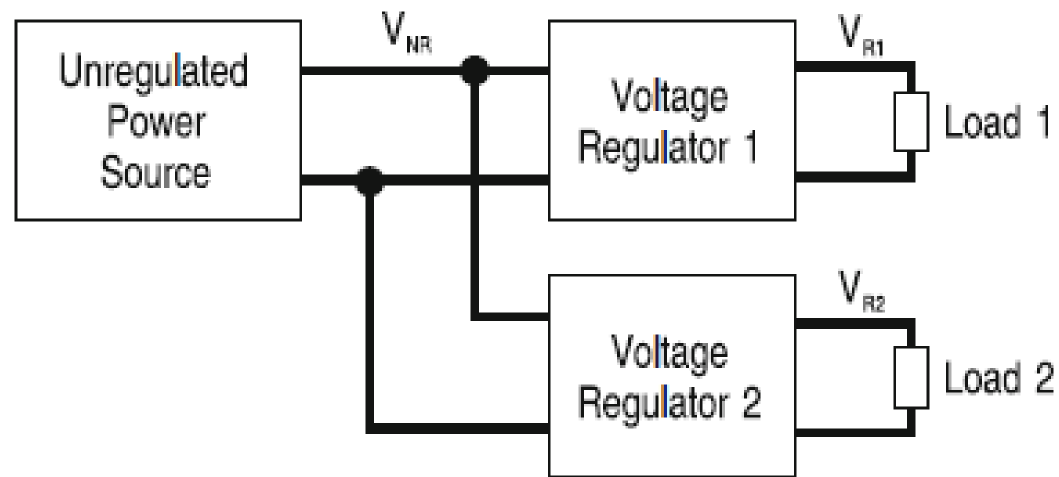


Figure. Structure of a typical MCU power supply

Bypassing Techniques

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**.

The **size** of a bypassing capacitor is selected in accordance to the frequency of the noise they are intended to attenuate. Assuming a supply voltage variation V_{cc}, a supply current I_{cc}, and a worst-case transient time T are known, the value for a bypass capacitor C_b can be estimated as $C_b = \frac{I_{cc}}{\Delta V_{cc/T}}$

In most cases, a standard non-electrolytic capacitor between 1 and 0.01µF shall work.

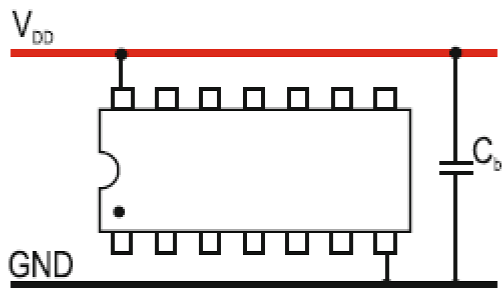


Fig. Placement of a bypass capacitor as close as possible to VDD–GND terminals **(location)**

Source Decoupling

Decoupling refers to the isolation of two circuits in the same path or connection line. Power supply decoupling is usually achieved through the installation of low-pass filters in strategic points of the power distribution line to reduce the strength of high-frequency noise components from one side of the system, reaching potentially sensitive components in the other side.

- In most cases a 10–100 μF electrolytic capacitor shall suffice.
- The value of the inductor, although not critical, is also recommended to be fairly large, with typical values in the range from 10 to 100mH.

In circuits sensitive to power supply ringing, the decoupling circuit could make the power supply underdamped.

If this were an issue, a damping resistor of value $R_{damp} = 2\sqrt{\frac{L}{C}}$

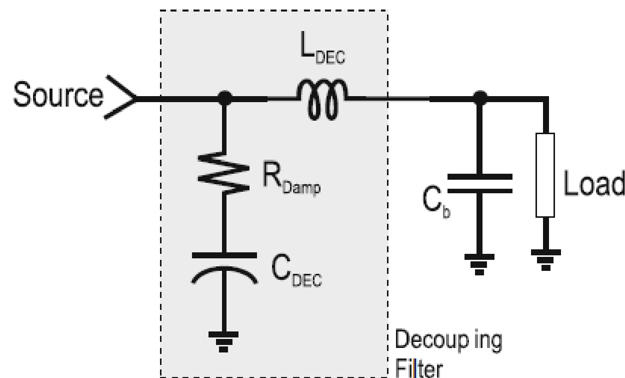


Figure. Typical topology of a decoupling LC filter including an optional damping resistor

Clock Sources

The need for a clock source in microprocessor-based systems arises from the synchronous sequential nature of the digital logic making up most of the system components.

The synchronous operation of the control unit's finite state machine (FSM) requires a periodic signal to yield precise transitions between the different states assumed by the CPU.

In many cases, particularly in MCU-based designs, the clock sources for both the CPU and peripheral devices are derived from the same clock generator, which is designated for that reason System Clock.

Figure. A 12 MHz Clock Source (Crystal Oscillator, XTAL)

Power-On Reset (POR)

The sequential nature of the CPU's control unit, besides a clock, also requires a RESET signal for taking its finite state machine (FSM) to its initial state upon power-up. All sequential circuits have such a requirement. The reset signal in a microprocessor-based system causes several actions in the CPU and its surrounding logic.

Figure. MSP430 LaunchPad and Reset Button (RST)

Bootstrap Sequence

The bootstrap sequence, frequently shorthanded as the Boot Sequence refers to the sequence of instructions that a CPU executes upon reset. What does it have to deal with boots and straps? The term bootstrap seems to be a metaphor of the phrase “pull oneself up by one’s bootstraps” denoting those with the ability of helping themselves without external means.

In a sense, that is what the bootstrap sequence does for a computing system: starting it when no other software is yet running. No basic microprocessor or MCU interface is complete without a properly designed boot sequence. (Similar to Firmware in various hardware and BIOS in the computer systems)

Bootstrap Sequence

Specific tasks performed in an MCU boot sequence are influenced by multiple factors that include: specific processor used, system architecture, application, and even the programming style. However, it is possible to identify typical operations taking place in this sequence.

- Identifying and initializing system peripherals.
- Setting-up the stack.
- Initializing system-wide variables in memory.
- Performing diagnostics and system integrity check-up.
- Loading an operating system or other type program.

MSP430 LaunchPad Block Diagram

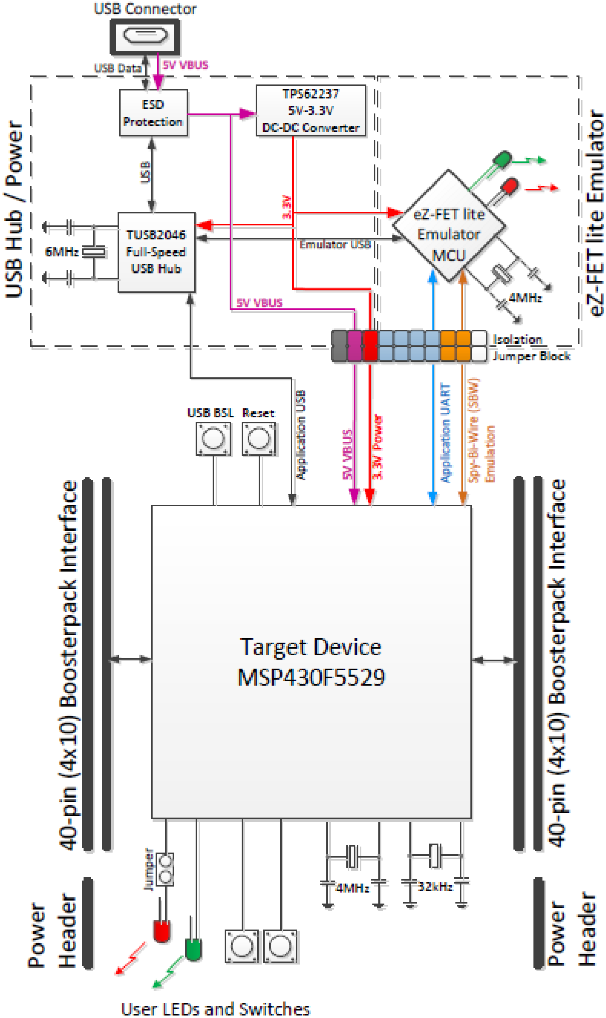


Figure. MSP430 LaunchPad Block Diagram

EEE 204

C Programming

Asst. Prof. Dr Mahmut AYKAÇ

Introduction to C Programming

All C programs start with a header…

```
#include <msp430.h>; //include all MCUs in MSP430 Family
#include <msp430x11x1.h>; //include a specific MCU
```

While the first description includes all the MCUs belongings to MSP430 family, second one includes a specific one.

Similar to standard C language, all programs must have main function and the entire program must be in it.

```
int main(void)
{ //Program starts here
.
.
}//ends here
```

Registers in C Programming

Registers (PxDIR, PxIN, PxOUT etc.) are same as they are in Assembly programming but to load the data to the related registers are different.

```
P3DIR=0xFF;  //All bits of Port 3 are output

P3OUT=0xAB; //Send AB (10101011) to Port3. P3.7, P3.5, P3.3, P3.1, P3.0 are ON, others are OFF

P2DIR=0xEE; //P2DIR=EEH=11101110, P2.0 and P2.4 are inputs, others are outputs
```

- In Assembly language, when we need to make a comment for each line of instruction we must use ";" but "//" in C.
- In Assembly language, no lines of instructions require ";" to indicate the ending point of line of instruction. But C requires ";". Otherwise it yields a syntax error. Recall from C or C++ lectures, control flow statements such as **`if`**, **`while`**, **`for`** etc. don't require ";". **Same also for programming our MCU in C**

Bitwise in C Programming

It also allows the user to use bitwise. Here are some examples…

```
P1OUT |= BIT4;  //Set P1.4
P1OUT ^= BIT4;  //Toggle P1.4
P1OUT &= BIT4; //Clear P1.4
```

Toggle means the corresponding PIN is 1 or 0 alternatively in each time of execution.

Watchdog Timer

A watchdog timer (WDT) is a timer that monitors microcontroller (MCU) programs to see if they are out of control or have stopped operating. It acts as a “watchdog” watching over MCU operation.

The watchdog timer communicates with the MCU at a set interval. If the MCU does not output a signal, outputs too many signals or outputs signals that differ from a predetermined pattern, the timer determines that the MCU is malfunctioning, sends a reset signal to the MCU and MCU restarts WDT for further need.

The operation of the watchdog is controlled by the 16-bit register WDTCTL.

Due to the performance and power issues, it is recommended to stop it at the beginning of code.

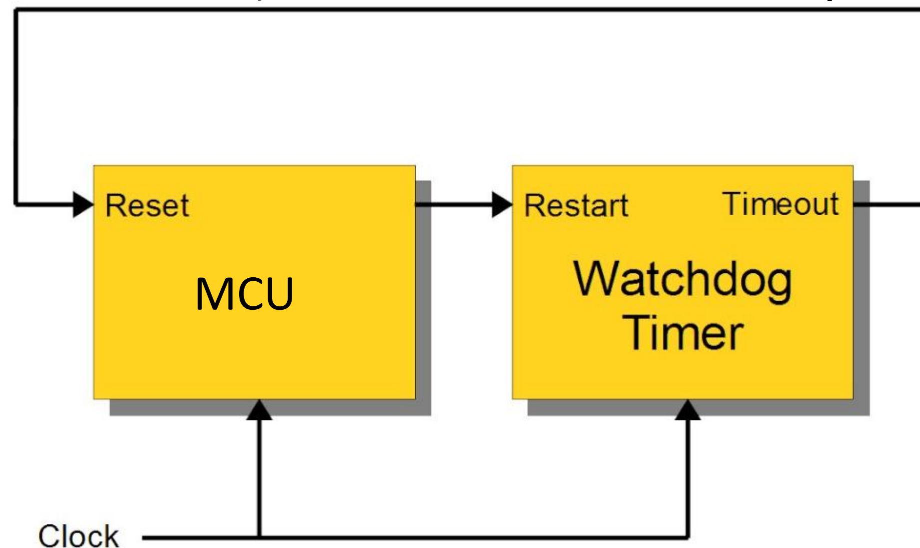


Figure. Watchdog Timer Mechanism

C Programming

Ex: Run the following code and observe that P4.7 is ON and OFF continuously.

```
#include <msp430.h>
int main(void)
{// start of main
WDTCTL = WDTPW | WDTHOLD; //stop watchdog timer

P4DIR=0xFF; //Set P4 as output
P4OUT=0x00; //Clear P4, recommended
while(1) //Infinite loop. Otherwise, program ends after 1 time of run
{
P4OUT ^= BIT7; //Toggle P4.7
}
return 0;
}// end of main
```

C Programming

We can rewrite the same program in a different way…

```
#include <msp430.h>
int main(void)
{// start of main

WDTCTL = WDTPW | WDTHOLD; //stop watchdog timer

P4DIR=0xFF; //Set P4 as output
P4OUT=0x00; //Clear P4
for(;;) //Infinite loop, otherwise program ends after 1 time of run
{
P4OUT = 0x80; //80H=1000 000, Therefore, P4.7 is ON
P4OUT = 0x00; //00H=0000 000, Therefore, P4.7 is OFF and continue, which means toggle
}
return 0;
} // end of main
```

Define Directive in C Programming

Just like C/C++, it is also possible to use **define** to name constants with a catchy way. Here are some examples…

```
#define LED1  BIT7 //Defining the 7th bit as LED1
#define LED2  BIT4 //Defining the 4th bit as LED2
#define BTN1  BIT1 //Defining the 1st bit as Button 1
#define BTN2  BIT3 //Defining the 3rd bit as Button 2
```

As seen from the examples given above, even though the bit order is defined, port numbers are not included. This gives us flexibility to use the same bit number on different ports without extra code.

Extra code causes extra load on CPU performance.

In contrast to Assembly, C is a case-sensitive programming language

Define Directive in C Programming

```
#include <msp430.h>
#define LED  BIT7 //Defining the 7th bit as LED
int main(void)
{
WDTCTL = WDTPW | WDTHOLD; //Stop watchdog timer
P4DIR=0xFF; //All bits of P4 are outputs
P4OUT=0x00; //Clear P4 for transferring data
P3DIR=0xFF; //All bits of P3 are outputs
P3OUT=0x00; //Clear P3 for transferring data
P5DIR=0xFF; //All bits of P5 are outputs
P5OUT=0x00; //Clear P5 for transferring data

while(1)
{
P4OUT ^= LED; //Toggle LED on P4.7
P3OUT ^= LED; //Toggle LED on P3.7
P5OUT ^= LED; //Toggle LED on P5.7
}
return 0;
}
```

Functions in C Programming

"A *function* is a block of code which only runs when it is called (invoked)".

Recall from C++ lecture, a function is invoked with or without parameters (depending on the function type), and the function returns to the invoking point with or without a value (depending on the function type).

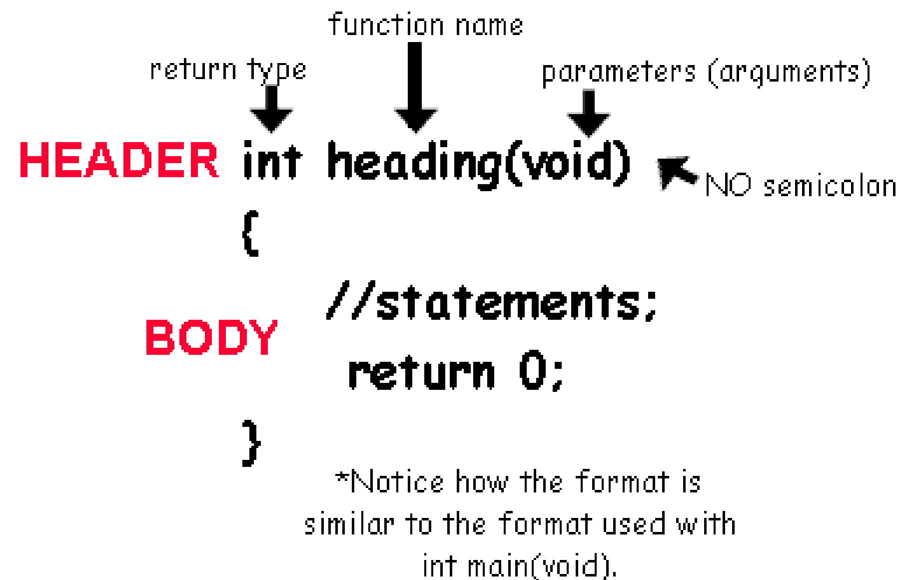


Figure. Definition of a function

Example for Functions in C Programming

Even though toggling means to turn ON and OFF, it is not possible to observe it without a reasonable delay.

Ex: Same program with some delay…

```
#include <msp430.h>
void MyDelay(void); //Declaration of the functions
void MyDelay(void) //Function starts here
    {
        volatile unsigned long int i; // i is 32 bit, unsigned and volatile(stored in RAM)
        for(i=1; i<25000; i++); //Killing time
    } //Function ends here
int main(void)
{
WDTCTL = WDTPW | WDTHOLD;// stop watchdog timer
P4DIR=0xFF; //P4 is out
P4OUT=0x00; //Clear P4
while(1) //Infinite loop
{
P4OUT ^= BIT7;   //Toggle P4.7
MyDelay(); //Call MyDelay function
}
return 0;
}
```

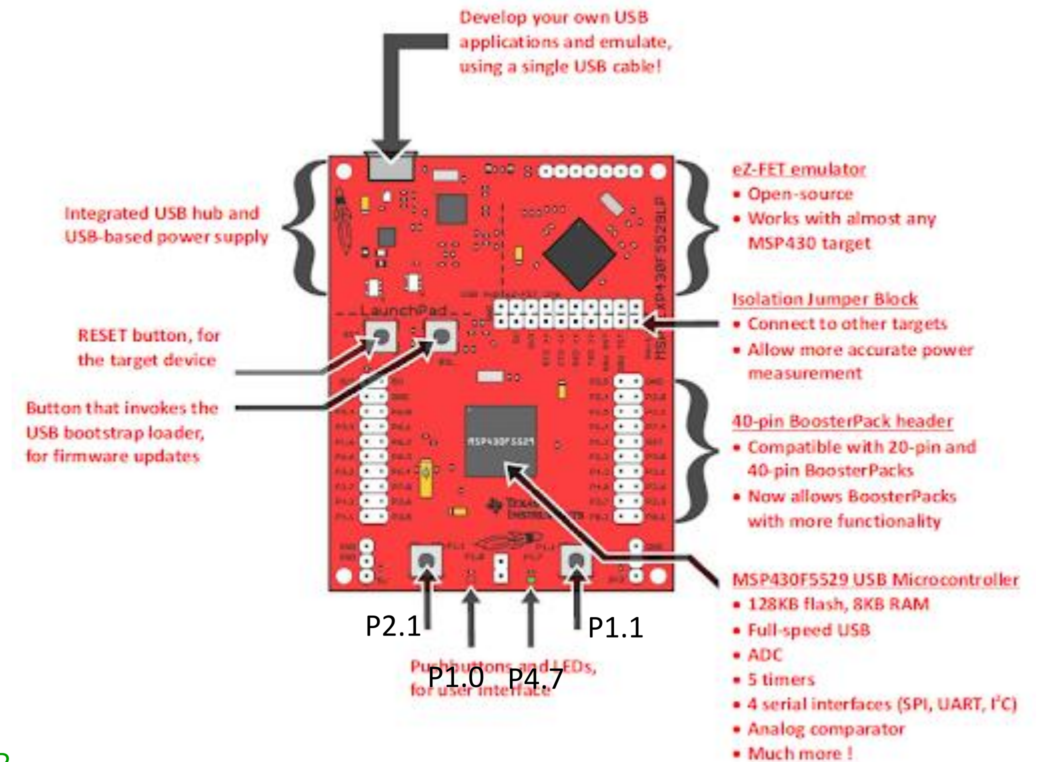
BEWARE!!! Even though the program flow **can** be through a label without a jump instruction in Assembly language, program flow **cannot** be through a function without an invoking instruction in C language.

More Examples in C Programming

```c
#include <msp430.h>
void MyDelay(void);
#define LED1      BIT7 //Defining the 7th bit as LED1
#define LED2      BIT0 //Defining the 1st bit as LED2
void MyDelay(void)
    {
        volatile unsigned long int i;
        for(i=1; i<25000; i++); //Changing the ending value also changes the amount of delay
    }
int main(void)
{
WDTCTL = WDTPW | WDTHOLD; //stop watchdog timer
P4DIR=0xFF;
P4OUT=0x00;
P1DIR=0xFF;
P1OUT=0x00;
while(1)
{
P4OUT ^= LED1;  //Toggle P4.7, LED1=P4.7
P1OUT ^= LED2;  //Toggle P1.0, LED2=P1.0
MyDelay();
}
return 0;
}
```

More Examples in C Programming

```
#include <msp430.h>
#define BUTTON   P2IN
#define LED      P4OUT
void MyLongDelay(void) //Long Delay Function
    {
        volatile unsigned long int i;
        for(i=1; i<25000; i++);
    }
void MyShortDelay(void) //Short Delay Function
    {
        volatile unsigned long int j;
        for(j=1; j<12500; j++);
    }
int main(void)
{
WDTCTL = WDTPW | WDTHOLD; //stop watchdog timer
P2DIR=0x00; //P2 is input
P4DIR=0xFF; //P4 is output
while(1) //Always check!, otherwise it checks once and ends the program
{
    while (BUTTON!=0xFD) //If button is not pressed, it is connected to P...
    {
        LED ^= BIT7;
        MyShortDelay();
    }
    LED ^= BIT7;
    MyLongDelay();
}
return 0;
}
```



Remember our MSP430 LaunchPad board has 2 LEDs connected to P1.0 and P4.7 while it has 2 buttons connected to P2.1 and P1.1 whose default states are logical 1.