ELECTRONICS CLUB

MICROCONTROLLER CODING

AN INTRODUCTION TO LAUNCHPAD

Meet the MSP430 Value Line
LaunchPad Development Kit
Complete Development Kit for $4.30
Supports MSP430 Value Line microcontrollers
Plug-in BoosterPack expansions available
On-board Flash Emulation Tool for programming/debugging
Everything you need to get started today!

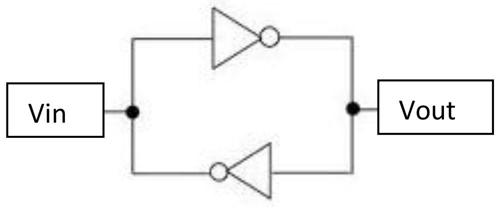
CFI, IIT MADRAS

Contents

Introduction to Microcontrollers

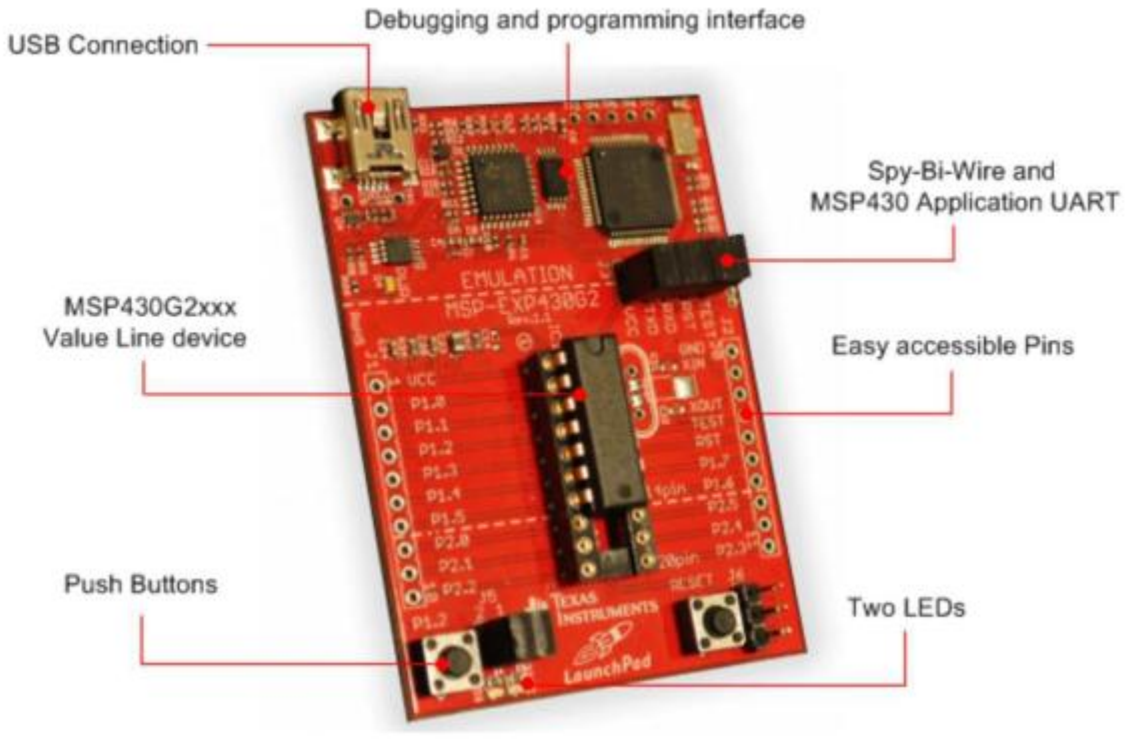
Before we all take the plunge, a few definitions:

1. **Microprocessor:** A microprocessor incorporates the functions of a computer's central processing unit (CPU). It is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output.
2. **Peripherals** : Digital/Analog blocks which do particular tasks. For e.g. ADC, Timers, etc.
3. **Microcontroller:** A microprocessor and its peripherals
4. **Register** : Smallest independent building-block which can store information. Registers are made up of flip-flops. The information stored by the registers can be used to control the outputs of peripherals (Eg. Counter limit of a Timer) or read the status of an external signal.
5. **Flip-flops** : Flip-flops can store 1 bit of information. A crude example of a flip-flop is given below:



6. **Architecture** : Architecture of a microprocessor defines a lot of things about the processor. It lays down the layout and the positioning of the components. It also decides on the instruction decoder procedure. Most of the modern day microprocessors follow RISC architecture. RISC stands Reduced Instruction Set Computer as opposed to CISC architecture which was a standard followed in the past. CISC architecture stands for Complex Instruction Set Computer.
7. **CPU** : CPU consists of a set of registers, program counter, ALU and the control unit. The registers in the CPU are the memory locations that the CPU have the quickest access to.
8. **Program counter:** Program counter, as the name suggests, keep track of the instruction line number.
9. **ALU** : ALU is the abbreviation for Arithmetic and Logic Unit.
10. **Control Unit** : Control Unit orchestrates the entire functioning of the microprocessor. It sends out command to fetch the instruction from memory, asks ALU to process it and decide based on the output, the future course of action.

Introduction to Launchpad


USB Connection
Debugging and programming interface
Spy-Bi-Wire and MSP430 Application UART
MSP430G2xxx Value Line device
Easy accessible Pins
Push Buttons
Two LEDs

The MSP-EXP430G2 low-cost experimenter board called **LaunchPad** is a complete development solution from the new Texas Instruments MSP430G2xx series. Its integrated USB-based emulator offers all the hardware and software necessary to develop applications for all MSP430G2xx devices. The LaunchPad has an integrated DIP target socket that supports up to 20 pins, allowing MSP430 Value Line devices to be used with the board comfortably. It also offers an on-board flash emulation tool allowing direct interfacing with a computer for easy programming, debugging and evaluation. It also provides a 9600 baud UART (Universal Asynchronous Receive Transmit) connection.

The chip we are going to use is **MSP430G2231** and has the following features:

1. Low supply Voltage: 1.8 V to 3.6 V
2. 16-bit RISC architecture
3. 16-bit Timer
4. 8 GPIO pins
5. 10-bit 200 kbps ADC
6. 2 kB Flash Memory
7. 512 B RAM

Writing a blink code

In the embedded systems world, blinking a LED is equivalent to the "Hello World" code in the world of programming languages.

As mentioned before, a microcontroller has general input/output pins (GPIO). In our case, we have around 8 GPIOs. These GPIO are multiplexed with the input/output of other peripherals in the microcontroller too. Multiplexing is a technique when multiple things are connected to same pin and based on the control settings; functionality of the pin is selected.

Problem Statement: There are two built-in LEDs in Launchpad. Write a code to get both the LEDs to blink alternately after a fixed interval, say 1 sec.

General Input-Output Pins

In MSP430G2231, there is a single port, **PORT 1.**

Three registers are associated to every port 'x' of any microcontroller:

1. PxDIR
    - Is a read/write register.
    - Decides the direction of the pin.
    - Its a 8-bit register and each bit in this register correspond to 8 pins starting from 0.
    - **Writing 1 to any bit sets the corresponding pin as an output pin.**
    - **Writing 0 (Clearing) to any bit sets the corresponding pin as an input pin.**
    - E.g.: A statement

      P1DIR=0b01000001

      sets pins 0 and 6 as output pins and the remaining pins as input pins.
2. PxOUT
    - Is a read/write register.
    - Decides the output of the pin, i.e., is it HIGH or LOW.
    - **Writing 1 to any bit pulls up the corresponding pin and makes it HIGH.**
    - **Writing 0 (Clearing) to any bit pulls down the corresponding pin and makes it LOW.**
3. PxIN
    - Is a read register
    - Contains the latest information on status of the input pins.
    - **Reading 1 from any bit means the corresponding pin is HIGH.**
    - **Reading 0 from any bit means the corresponding pin is LOW.**

Writing numbers in Computers

Any number, by default, is considered as a decimal number. A prefix of '0b' tells the instruction decoder that it is a binary number and a prefix of '0x' marks it hexadecimal.
$(17)_{10}$=0b10001=0x101

Some pointers regarding the syntax followed

Even though most of the embedded systems are programmed in languages like C, since the programming involves bit and register manipulations, some interesting tricks are used to make life simple.

1. To set pin 6 high, we do

   P1OUT=0x40 or *P1OUT=0b01000000* or *P1OUT=64*

   - Readability, in the above statement except for the binary case, isn't very good. But the binary way of assigning is extremely prone to mistakes.
   - A way out is use of left-shift operators (<<)

   P1OUT=(1<<6)

     - (1<<x) can be interpreted as 0b1 being shifted left 6 times. Hence (1<<6)=0b01000000.
     - (1<<0) is 0b1.
     - Analogously, we have right-shift operator.

2. To set pins 6 and 4 high, we do

   P1OUT=0x50 or P1OUT=(1<<6)|(1<<4)

   - '|' is bitwise OR. So (0b01000000)|((0b00010000)=0b01010000 which is nothing but 0x50.
   - Truth table of OR

X	Y	X|Y
0	0	0
0	1	1
1	0	1
1	1	1

3. To set pins 6 and 4 without affecting other pins, we do

   P1OUT=P1OUT | 0x50 or P1OUT= P1OUT | (1<<6)|(1<<4)

   OR

   P1OUT |= 0x50 or P1OUT |= (1<<6)|(1<<4)

4. To clear pin 6 and 4 without affecting other pin settings, we do

   P1OUT &=~0x50 or P1OUT &=~((1<<6)|(1<<4))

   - '&' is bitwise AND. So (0b01000000)&((0b00010000)=0b00000000.
   - Truth table of AND

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

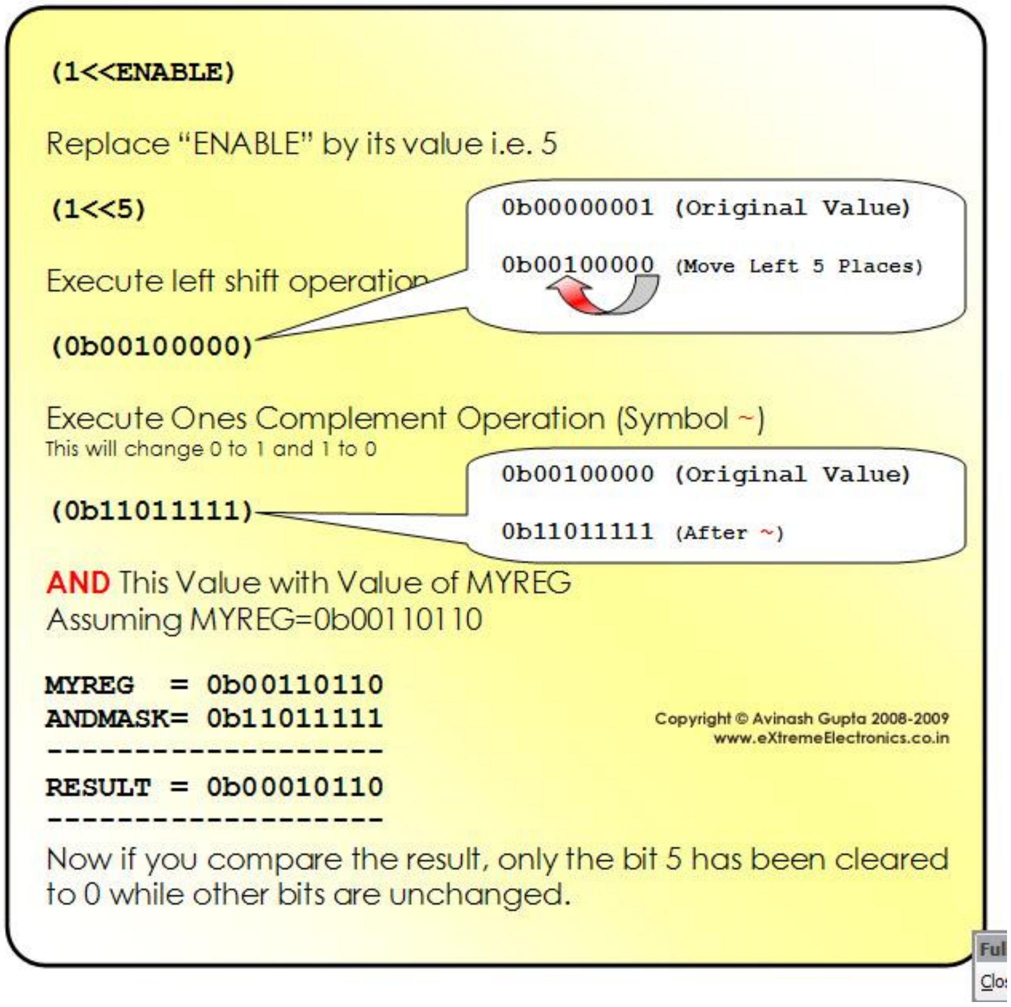
   - '~' is bitwise complement (NOT). So ~(0b01000000)=0b10111111

- Truth table of NOT

X	~X or !X
0	1
1	0

- Hence, as you can see, ~0x50=0b10101111. ANDing any register with this clears pin 6 and pin 4 of the register.

An illustration below might help drive this slightly complex point home. Consider the statement MYREG &=~(1<<ENABLE), where ENABLE was defined to be 5.



(1<<ENABLE)

Replace "ENABLE" by its value i.e. 5

(1<<5)

Execute left shift operation

```
0b00000001  (Original Value)
0b00100000  (Move Left 5 Places)
```

(0b00100000)

Execute Ones Complement Operation (Symbol ~)
This will change 0 to 1 and 1 to 0

```
0b00100000  (Original Value)
0b11011111  (After ~)
```

(0b11011111)

AND This Value with Value of MYREG
Assuming MYREG=0b00110110

```
MYREG   = 0b00110110
ANDMASK= 0b11011111
--------------------
RESULT  = 0b00010110
--------------------
```

Now if you compare the result, only the bit 5 has been cleared to 0 while other bits are unchanged.

Copyright © Avinash Gupta 2008-2009
www.eXtremeElectronics.co.in

5. To toggle pins 6 and 4 without affecting other pins, we do

P1OUT=P1OUT ^ 0x50 or P1OUT= P1OUT ^ (1<<6)|(1<<4)

OR

P1OUT ^= 0x50 or P1OUT ^= (1<<6)|(1<<4)

- Toggling is equivalent to complementing a single bit. For eg, switching an LED on if it is off and switching it off if it is on can be easily achieved using XOR operator (^).

- Truth table of XOR (Exclusive OR -> Equivalent to OR but 1^1=0)

X	Y	X ^ Y
0	0	0
0	1	1
1	0	1
1	1	0

  - Can be understood as saying X^Y is 1 iff no. of ones is odd.
- In our case, we want a specific signal X to be complemented. We realize from the truth table that X^0=X and X^1=complement of X.
- Thus, we can complement specific pins in our register according to our requirement by using a suitable mask.

Some code-specific pointers

1. Header files for this code would be
   #include "msp430.h"   // Has the macros for all the registers, e.g., P1OUT,etc.
2. We need to have some sort of delay function. Use the built-in function _delay_cycles(no. of cycles) for this. Remember the no. of cycles should be preferably within 2000 for accuracy.
3. TI's modules come with a built-in watchdog. Use the following line of code to disable the watchdog.
   WDTCTL = WDTPW + WDTHOLD;   // Stop watchdog timer
4. Remember to have the code loop infinitely.

Relating to the real world….

The programming techniques you have learnt in this section is quite universal in nature and is used everywhere. To cite an example, the code that runs in your mobile processor is also written in C. The registers of your mobile's microprocessor are also initialized and manipulated using the above techniques. Hence, please do make sure that you have understood each and every technique.

Using IAR Embedded Workbench Kickstart

IAR Embedded Workbench Kickstart is provided by Texas Instruments as an Integrated Development Environment for developing codes on MSP430 series. In this section, we will explore this software a bit and understand how to use it to download our codes on the microcontroller.

Go through the following steps carefully:

1. Create a folder on the desktop and name it as you wish. I will be referring to it as XYZ.
2. Open IAR Embedded Workbench Kickstart software.
3. Open a new workspace and save it inside this folder. Name it as you wish.
4. Open a new project inside this.
    - Go to projects> Create a new project
    - Make it an empty project
    - Save it inside XYZ and name it as you wish. I will be referring to it as PRO.
5. In the workspace tab on the left, change the nature of the project from Debug->Release.
6. Right-click on the project PRO-Release and click options.
    - Inside General Options,
        i. Target
            - Device -> MSP430G2231
                - Use Drop-down menus to guide yourself to the correct microcontroller.
                - As it is obvious, should you ever use another device, this is where you need to mention the chip number.
    - Inside Linker
        i. Output
            - Format
                a. Check Debug information for C-SPY and the subsequent 2 check-boxes.
    - Inside Debugger
        i. Driver
            - FET Debugger
7. Now add the file blink.c into this project by right clicking PRO once again.
    - In the menu that dropped down, click on Add> Add Files
8. Your project is ready now.
9. Build the project by clicking Project>Build and if its a re-run then click Project> Rebuild All.
10. Load the project onto your Launchpad by clicking Project> Download and Debug.
11. Press F5.

Congratulations!!! You have got your blinky code working. :)

Relating to the real world....

Using the above knowledge, you can design your own Christmas Lighting. Yes, the whole thing sounds trivial but getting the LEDs going on a pattern designed by yourself will give you an unmatched joy. The following might be something you may already know but still mentioned here for closure:

1. LED is a light-emitting diode. It generates light when voltage is applied across it. Its current-voltage characteristics are given in Fig a. Since higher voltages imply huge currents, we put a resistor, generally 330 ohm, in series with the diode. Fig b shows a typical LED with the leads named.

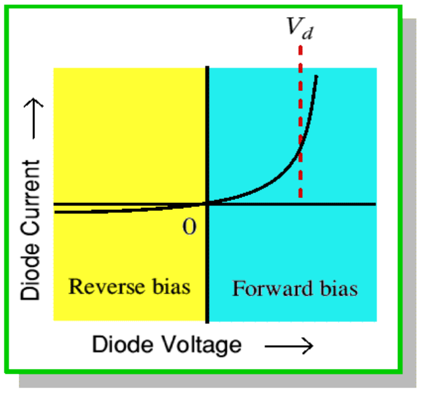


Fig a

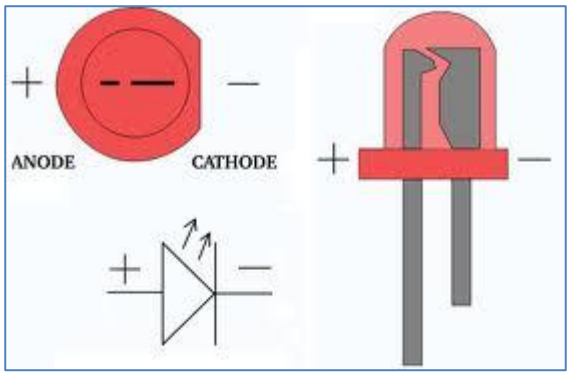


Fig b

2. Resistors follow the colour coded scheme for easy identification. A famous mnemonic is B B ROY of Great Britain had a Very Good Wife.

Using the switch button

Now, we move to taking inputs and manipulating the flow of control in the code using them. Launchpad comes with a switch onboard. This switch is active low and hence when not active gives HIGH (3.3 V) and when activated gives LOW (0 V).

Problem Statement:
Write a code to do the following using the switch and LEDs present on the Launchpad:

1. If the switch is not pressed, turn on LED1.
2. If the switch is pressed, turn on LED2.

Some code-specific pointers

1. To know the state of the switch, use the following lines

```
if(P1IN & (1<<3))
{
.......................
}
```

Do remember that everything in C programming carries forward here. That is, you can use if-else, switch case, while loops, etc. You can go ahead and use even pointers.

Relating to the real world....

Using the above knowledge, you can design your own people counter, the circuit we demonstrated in our first session. Use lasers as the sensor and feed the output of the sensor into a GPIO pin and increment/decrement a global variable 'i' as when the situation demands. For the display, use a BCD-7 segment display IC (7447) which can help you display 'i' with the help of just 4 pins.

Using Interrupts

Problem Statement:
Write a code to do the following using the switch and LEDs present on the Launchpad:

1. If the switch is not pressed, LED1 and LED2 turn on alternatively after a time interval, say 1s.
2. If the switch is pressed, LED1 and LED2 turn on together after a time interval, say 1s.
3. The transition must happen immediately.

The above problem statement can't be achieved using polling, the formal name for the technique we had used in our previous task. Can you think of way you can do it? Our biggest problem is the delay_cycles function which we have take the control away from us. While that function is being executed we can't read the input, which causes all the headache.

Interrupts help us achieve partial parallelism in low-level devices. It increases our productivity by helping CPU focus on multiple things with minimal resources.

Consider the case, an interesting cricket match is going on TV. Unfortunately, you have a class but you are not sure if the professor will come. What would you do?

A not-so-intelligent person X would go to class. If the professor comes, then X would sit through the class. But if the professor doesn't come, then X would have to come back to the hostel's common room to watch the remaining of the match. Meanwhile, X would have missed some important moments of the match.
A clever person Y would have sent a friend to the class. If the professor comes, then the friend would call Y and Y would rush to class. But if professor doesn't come, then Y wouldn't have missed any part of the match.

As you can see, the productivity has increased in the second option because of delegation. Delegation helps in optimizing the use of resources. But, delegation requires management of multiple tasks at the same time. Interrupts to come to our rescue during this situation.

Interrupts

On receiving an interrupt, the CPU saves the current local registers and the program counter to a stack and loads the interrupt service routine (ISR). This ISR helps modify the flow of control in the normal code and thereby introduce a partial multi-tasking into the system. After the execution of ISR, the CPU restores the local register and the program counter and resumes with whatever task it was doing while the interrupt arrived.

In a microcontroller's program memory, every interrupt is allocated a specific space for the interrupt service routine. The starting memory address of each code section is referred to as an interrupt vector. While there are several interrupt service vectors available in the microcontroller, the one relevant to us is in this case is PORT1_VECTOR. Others include TIMERA_VECTOR, etc. They can be found in the datasheet.

To use the interrupts associated with the pins of a port we need to set values in three registers:

1. P1IE
    - Is a read/write register.
    - Enables the interrupt associated with the GPIOs
2. P1IES
    - Is a read/write register.
    - Decides which transition in the selected pins triggers the interrupt
    - **Writing 1 to this causes interrupt to be triggered when this point transits from HIGH to LOW**
    - **Writing 0 to this causes interrupt to be triggered when this point transits from LOW to HIGH**
3. P1IFG
    - Is a read register
    - Whenever a pin triggers an interrupt, the corresponding pin in P1IFG is set to 1. Essentially, this register acts as the flag for CPU to watch out for interrupts.
    - Needs to be cleared by the software after finishing the execution of ISR.

Some code-specific pointers

1. Header files for this code would be

    #include "msp430.h"   // Has the macros for all the registers, e.g., P1OUT,etc.
2. For the ISR, the following syntax is followed:

    #pragma vector=PORT1_VECTOR

    __interrupt void Port_1(void)

    The #pragma vector directive is used for specifying the interrupt vector address, in this case the interrupt vector for the PORT1 interrupt, and the keyword __interrupt is used for directing the compiler to use the calling convention needed for an interrupt function.
3. To enable the interrupt, after definition of the interrupt settings, include the following line:

    __BIS_SR(GIE);

SR is the status register of the microcontroller and __BIS_SR sets the corresponding pin to general interrupt enable in SR as specified by the macro GIE.

4. Since space allocated for ISR is small and the actual flow in the code is halted, always ensure that ISR is kept as small as possible. Use ISR to divert the flow instead of obstructing it.

Relating to the real world....

Consider the following additional requirements for our indigeniously created people counter:

1. A temperature monitor which monitors the temperature of the room in regular intervals
2. A set of switches which allows you to type in specific commands, like lock the door, reset the counter, etc.

As you must have noticed, we are essentially asking our poor microcontroller to do multiple tasks at the same time. It has to keep track of people, monitor temperature and also poll the keypad matrix. Since a polling-based approach to this problem is going to be very difficult to implement, we need to resort to using interrupts. We could assign interrupts to the switches as well as the the sensors. Inside our while(1) loop, we cn have the temperature sensor's value been read out periodically.

Timers and PWM

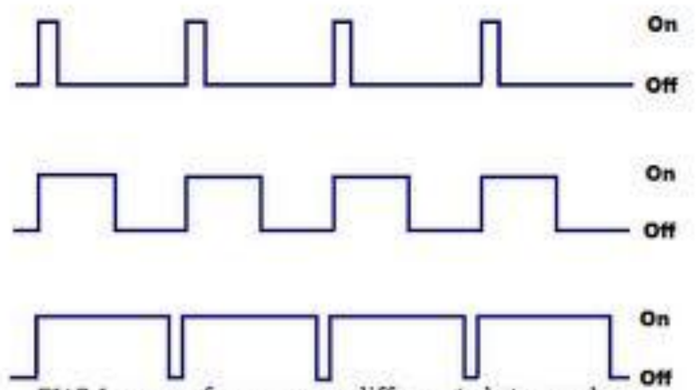
Problem Statement:
Write a code to do the following using LEDs present on the Launchpad:
1. Toggle LED1 using the internal clock of the uC
2. Adjust brightness of LED1 using PWM

Brightness of an LED can be adjusted by the current flowing through it, which in turn can be controlled by the voltage across its ends. Clearly, a '0' or '1' give only 'off' or 'on'. But if we can vary the period inside a cycle in which the signal is HIGH and LOW, we can, in principle, derive an analog signal from a digital circuit.

We define duty-cycle as the fraction of the cycle in which a signal is HIGH.

Timers and Pulse Width Modulation

The average (DC) value of the signal gives us an analog emulation. The higher the duty cycle, higher is the DC value. This works in the case of LED brightness control because the switching frequency is way higher than our persistance of vision.



PWM: same frequency, different duty cycles

But to implement PWM, a clock is needed for the following reasons:
- Adjustable period of PWM
- Adjustable duty cyle

All uCs come with their own internal clocks, that can be left to tick automatically or programatically.

Some code-specific pointers

The following registers are useful in manipulating clocks:

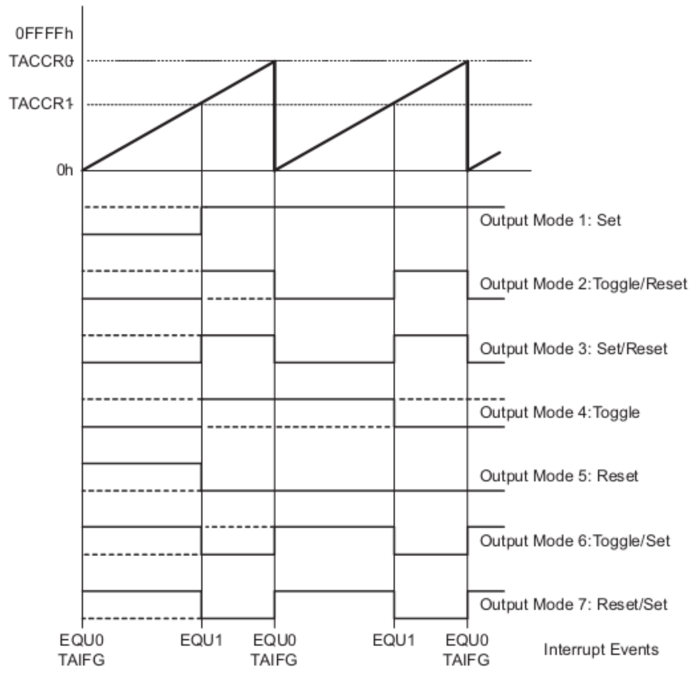
Register	Short Form	Register Type	Address	Initial State
Timer_A control	TACTL	Read/write	0160h	Reset with POR
Timer_A counter	TAR	Read/write	0170h	Reset with POR
Timer_A capture/compare control 0	TACCTL0	Read/write	0162h	Reset with POR
Timer_A capture/compare 0	TACCR0	Read/write	0172h	Reset with POR
Timer_A capture/compare control 1	TACCTL1	Read/write	0164h	Reset with POR
Timer_A capture/compare 1	TACCR1	Read/write	0174h	Reset with POR
Timer_A interrupt vector	TAIV	Read only	012Eh	Reset with POR

**POR := Power-On Reset*

The Capture Control Registers (CCR) together with the TACTL register trigger interrupts in TAIV register.

For example: In UP mode (MC0 = 01h), the timer counts up to whatever is stored upto TACCR0, triggers an interrupt and falls to zero again, like a sawtooth. Check out TACTL register specifications in page 372 of the family guide for the various options with which we can control the timer A.

Once timers are understood, PWM is straightforward. For example, when in upmode, the behaviour is as follows:



Relating to the real world....

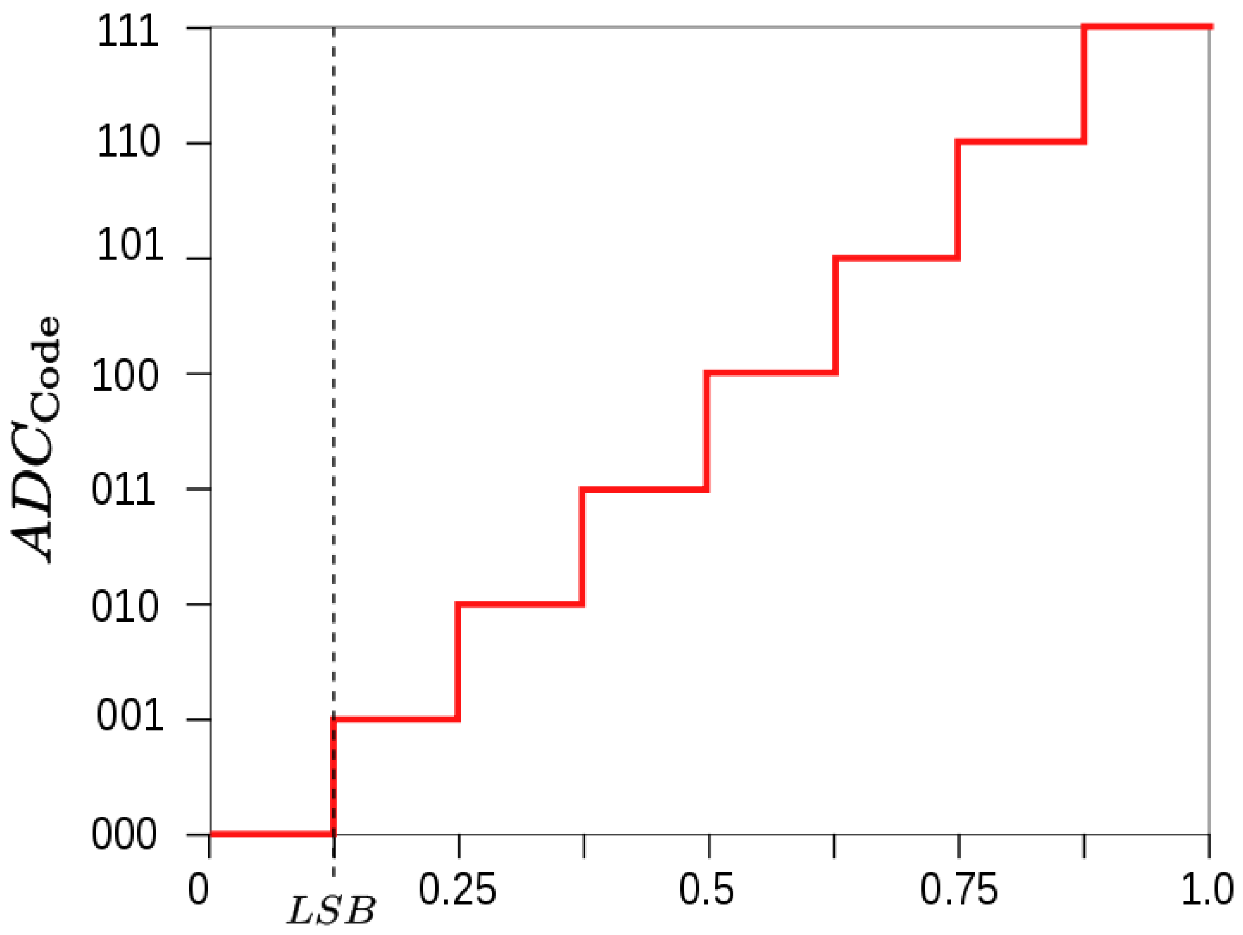
The principles discussed above finds its usage in lots of applications like mood lighting, driving motors, power efficient audio amplifiers (class D audio amplifiers), etc. Timers are used in all embedded systems; proof of which is that every uC has an inbuilt timer. PWM also helps derive analog signals from digital circuits. This will find great use in applications which require both analog and digital signals.

Analog to Digital Converters

Problem Statement:
Write a code to implement ADC using DC power supply and uC. Specifically, turn some LEDs on or off when the input voltage is above or below a certain threshold.

ADCs convert an input analog voltage to a digital number that represents the amplitude. Clearly, there would be a round-off error, aka quantization error. This error and the maximum frequency at which the ADC device can operate correctly are two aspects to keep in mind while designing.

A 3 bit ADC would behave as follows:



Some code-specific pointers

As was the case with Timers and PWM, the programmer needs to look into which registers have to be manipulated, ie, which bits to be set.

The registers involved are:

Register	Short Form	Register Type	Address	Initial State
ADC10 input enable register 0	ADC10AE0	Read/write	04Ah	Reset with POR
ADC10 input enable register 1	ADC10AE1	Read/write	04Bh	Reset with POR
ADC10 control register 0	ADC10CTL0	Read/write	01B0h	Reset with POR
ADC10 control register 1	ADC10CTL1	Read/write	01B2h	Reset with POR
ADC10 memory	ADC10MEM	Read	01B4h	Unchanged
ADC10 data transfer control register 0	ADC10DTC0	Read/write	048h	Reset with POR
ADC10 data transfer control register 1	ADC10DTC1	Read/write	049h	Reset with POR
ADC10 data transfer start address	ADC10SA	Read/write	01BCh	0200h with POR

Again, the control registers (CTL0 and CTL1) determine behaviour of the ADC (options of setting the minimum and maximum values of input analog voltage, for example).

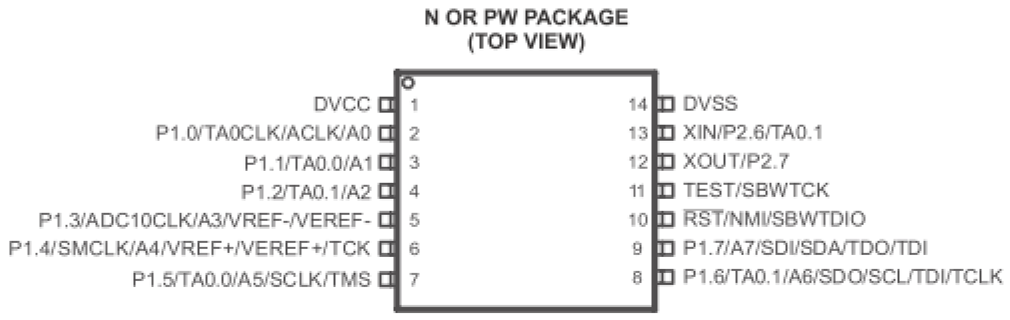
The exact bits for ADC10CTL1 and ADC10CTL0 can be retrieved from the user guide under the ADC10 section.

Once ADC is done for a value, with the help of the triggered interrupt, we can read in the digital value from memory (ADC10MEM).

Experimental Setup

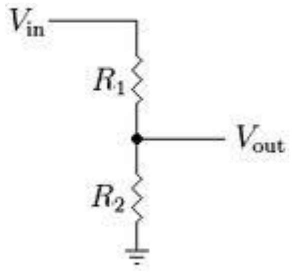
Since we use a DC power supply (range: 0-32V) for the input analog voltage, we might end up frying the microcontroller. Hence, connect the circuit in the breadboard as per following instructions:

1. The following is the pin-out of the microcontroller.



2. Connect:
    a. Pin 1 => 3.3 V (Obtain this from the port C of the SMPS. Adjust the voltage first and then draw +3.3V from the Red knob and ground from the black knob)
    b. Pin 14 => Gnd
    c. Pin 6 => Input for the analog signal.

i. If you are going to use the port A (0 – 30 V) of the SMPS, use a resistive divider to be safe. Connect the system as given in the diagram.



- Vin is from SMPS
- $\frac{R1}{R2} = 9$, this will ensure that Vout is 10 times smaller than Vin. Our input to the ADC must vary between 0-3.3V but Vin varies from 0 to 30 V.
- Also, resistors must be in the order of kilo ohms to reduce the current that drives into the ADC.

ii. If you are going to use the function generator, either make sure the amplitude voltage is set between 0 to 3.3V or use the resistive divider.

d. Pin 2,3 => Output for our logic.

i. We will be using the oscilloscope to detect the output. Please listen to our instructions carefully.

Relating to the real world....

The principle of ADCs has been demonstrated in this session but the applications are widespread ranging from use in Digital Storage Oscilloscopes to use in digitizing music.

References

Websites:

1. http://processors.wiki.ti.com/index.php/MSP430_LaunchPad_(MSP-EXP430G2)
    - Gives code snippets on a variety of peripherals available on MSP430 Value Line Series chips.
2. www.extremeelectronics.com
    - Gives a detailed explanation of various peripherals from an ATMega16 point of view. But using the Family Guide provided by Texas Instruments (TI), information obtained from this site can be easily applied to our chips.
3. www.msp430launchpad.com

Datasheets:

All these datasheets have been provided in the webpage too for easy access.

1. TI's Family Guide
    - TI uses the same architecture for the peripherals across all chips. Hence, they provide a single guide for all chips and the datasheet relevant to a chip gives us the information on what all are there in the chip under consideration.
2. MSP430G2231's datasheet

Appendix A: Writing a makefile, Compiling and Uploading the code

Now that we have written our code, we need to compile the same for errors. After removing the bugs, we can upload the code into the Launchpad using a software called mspdebug. The commands for compilation of the code and generation of the elf file are:

```
msp430-gcc -mmcu=msp430g2231 -O2 -Wall -c -o main.o main.c
msp430-gcc -mmcu=msp430g2231 -o main.elf main.o
```

The first command compiles and gives the object file which is used in the second command to generate the elf file. elf file (Executable and linking file) is then loaded on the Launchpad.

Makefile

In order to make our life easier and not have to sit and type the above commands again and again, we create a makefile which will do this task for us.

```
# makefile configuration
NAME          = main
OBJECTS       = ${NAME}.o
CPU           = msp430g2231

CFLAGS        = -mmcu=${CPU} -O2 -Wall

#switch the compiler (for the internal make rules)
CC            = msp430-gcc

#all should be the first target. it's built when make is run without args
all: ${NAME}.elf

#additional rules for files
${NAME}.elf: ${OBJECTS}
	${CC} -mmcu=${CPU} -o $@ ${OBJECTS}

clean:
	rm -f ${NAME}.elf ${NAME}.o

#project dependencies
${NAME}.o: ${NAME}.c
```

Uploading the code

We upload the elf file onto Launchpad using a software called mspdebug. The procedure for upload are as follows:

To execute the code step-by-step use the following command

(mspdebug) step

$mspdebug rf2500

(mspdebug) erase

Erases the memory

(mspdebug) prog xyzabcd.elf

Loads the code generated using xyzabcd.c

(mspdebug) run

Runs/resume the code

As the terminal reads, Ctrl+c will pause the code execution.