LaunchPad MSP430 Assembly Language Tutorial

http://robotics.hobbizine.com/asmlau.html

The Texas Instruments LaunchPad is a handy tool for evaluating and learning about the MSP430 Value Line series of microcontrollers. This tutorial uses the LaunchPad with its included MSP430G2231 processor to introduce MSP430 assembly language programming. A small program is developed which reads the status of a LaunchPad push button. While the button is not pushed the LaunchPad's red LED is turned on. If the button is pushed the green LED is turned on.

Although this first program is short - only 20 lines of code - the tutorial covers a lot of ground. In addition to an overview of the basic syntax and structure of MSP430 assembly language, information is provided on how to:

- Configure a pin as an input or output
- Turn an output on or off
- Read a digital input
- Branch to a different place in a program
- Implement a simple IF/THEN construct
- Implement a simple LOOP construct
- Implement a simple DELAY construct

The syntax presented here is based on TI's Code Composer Studio (CCS) and therefore to complete this exercise you will need to have CCS downloaded and installed on your computer. A free version of CCS is available from the TI website.

Basic Assembly Language Syntax

An assembly language source program is a text file made up of a series of source statements. A source statement will contain one or more of the following elements:

- Label - acts as a marker in the source code, useful as the destination in branching statements and subroutine calls
- Mnemonic - a machine instruction, assembler directive, or macro
- Operand List - one or more values to be operated on by the instruction, directive, or macro
- Comment - note inserted by the programmer that is not included in the assembly

The following example source statement contains all of these elements:

```
label       mnemonic        operand list            comment
Mainloop    bit.b           #00001000b,&P1IN        ; Read switch at P1.3
```

The most fundamental of the elements are the machine instruction mnemonics, as these map directly to the functions of the microcontroller's CPU. The complete MSP430 instruction set consists of 27 core instructions. There are an additional 24 emulated instructions which are translated to core instructions by the assembler software. When compared to the command set of some higher level languages, these machine instructions are very basic, performing tasks such as mathematical and logical operations, writing and retrieving values from memory, and branching to different sections of code.

For example, the instruction statement:

```
          add.w   #10,R5
```

takes the number 10, adds it to the value stored in general register R5 and stores the result back in R5.

While the instruction statement:

```
          jmp   Delay
```

will relocate program execution to the point marked by the Delay label.

Operands are very flexible. In the two examples above there are three different operand types - the number 10, the general register R5, and the user-defined symbol label Delay. In all, there are seven different addressing modes as well as support for numerous constant types including binary, decimal, hexadecimal, and ASCII.

The nature of assembler directives is very different from that of machine instructions. While machine instructions relate directly to the operation of the microcontroller itself, assembler directives are instructions related to the assembly process. A few common functions of directives include defining symbols, initializing memory, and specifying libraries from which the assembler can obtain macros.

Assembler directives can be distinguished from machine instructions by their leading dot, for example:

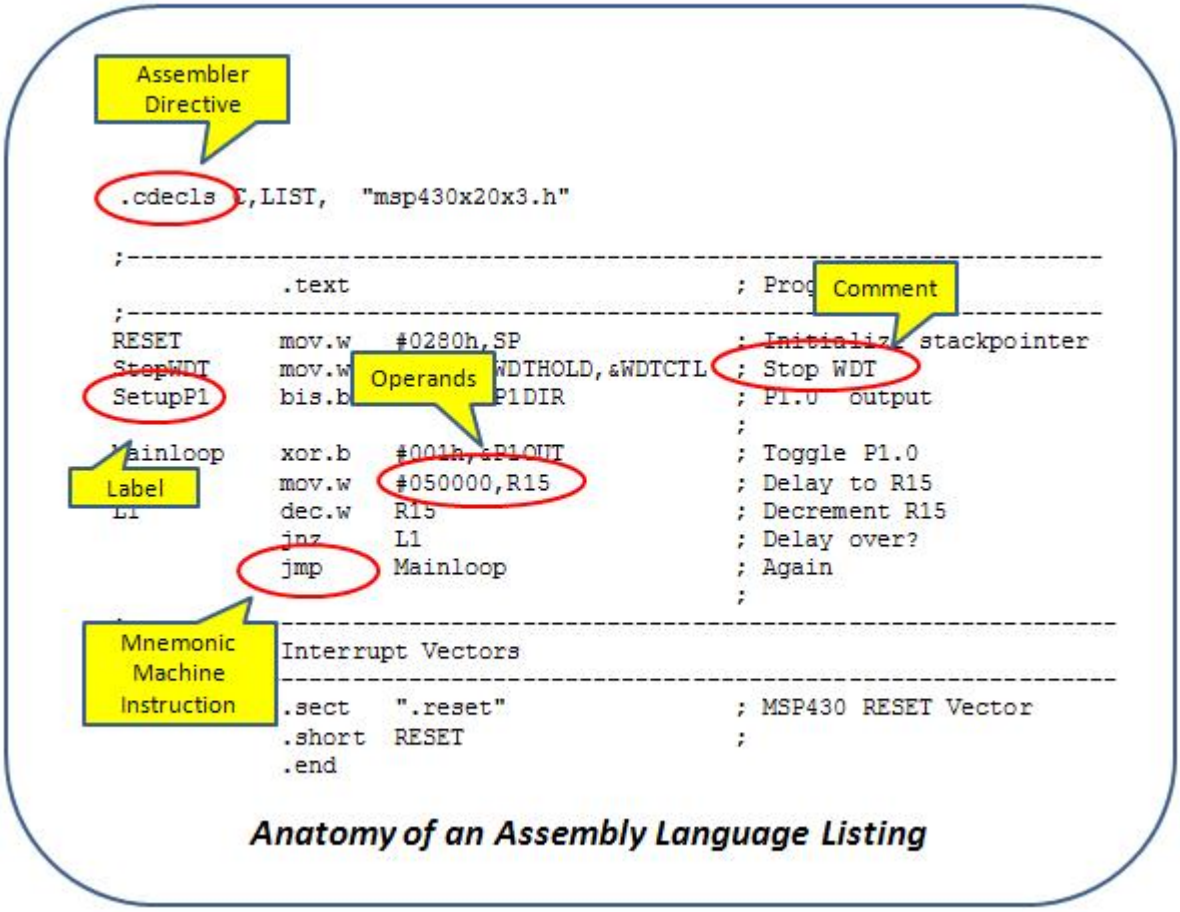
```
        .end
```

is the directive which tells the assembler that it has reached the last source statement of a program.

Labels are optional. When used they must be the first character of a line. They cannot begin with a number or contain a blank space, but otherwise can include the alphanumeric characters A-Z, a-z, 0-9, _, and $. Labels are case sensitive by default but can be set as case insensitive as an option. A label can be followed by a colon(:), but this is not required and the colon is not treated as part of the label name when referenced elsewhere in the program.

Comments are also optional. They can use any ASCII character including spaces. When a comment starts at the first character in a line it can begin with either an asterisk(*) or a semicolon(;), if it does not begin at the first character it can only begin with a semicolon.

The first character of a line can only be a label, asterisk, semicolon, or a blank space. The first character cannot be a mnemonic or it will be treated as a label by the assembler.



Anatomy of an Assembly Language Listing

A Note About Architecture

The register is an extremely important concept in microcontroller programming. On the MSP430 all operations except for program flow instructions are implemented as register operations. To set the frequency of a timer, a value is written to a register. To retrieve the status of an input pin, a value is read from a register. To save a value for later, that value is written to a register. In terms of the CPU all of these operations are functionally the same - a bit, byte, or word is written to or read from a specified location. Once you have mastered the basic techniques for manipulating these bits, bytes and words, learning to implement a feature of the MSP430 becomes largely a matter of using manuals and other resources to study the registers associated with that feature to figure out what configuration will produce the desired result.

The task of reading and writing to registers as discrete memory locations is greatly simplified by the use of symbolic aliases in the assembly program. Rather than using numeric addresses, the registers and bits are typically referenced by more or less meaningful names such as SP for "Stack Pointer" or P1OUT for the "Port 1 Output Register."

The Example Program

The example program for the tutorial appears below. As explained earlier, it is a simple program which is tied to the built-in hardware of the LaunchPad. If the button is pushed the green LED is on; if the button is not pushed the red LED is on. The code is fully commented and you will likely find it easy to follow the basic logic simply by reading the comments.

```
;-------------------------------------------------------------------------------
;   Digital I/O example for the LaunchPad
;   Read the status of built in push button - P1.3
;      (Note that P1.3 is "1" when the push button is open
;       and "0" when the button is closed)
;   Red light if the button is not pushed - P1.0
;   Green light if the button is pushed - P1.6
;   Build with Code Composer Studio
;-------------------------------------------------------------------------------

            .cdecls C,LIST,"msp430g2231.h"  ; cdecls tells assembler to allow
                                            ; the c header file

;-------------------------------------------------------------------------------
;   Main Code
;-------------------------------------------------------------------------------

            .text                           ; program start
            .global _main                 ; define entry point

_main       mov.w   #0280h,SP               ; initialize stack pointer
            mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; stop watchdog timer

            bis.b   #01000001b,&P1DIR       ; make P1.0 and P1.6 output
                                            ; all others are inputs by default

Mainloop    bit.b   #00001000b,&P1IN        ; read switch at P1.3
            jc      Off                     ; if P1.3 open branch to Off label

On          bic.b   #00000001b,&P1OUT       ; clear P1.0 (red off)
            bis.b   #01000000b,&P1OUT       ; set P1.6 (green on)
            jmp     Wait                    ; branch to a delay routine

Off         bis.b   #00000001b,&P1OUT       ; set P1.0 (red on)
            bic.b   #01000000b,&P1OUT       ; clear P1.6 (green off)

Wait        mov.w   #1834,R15               ; load R15 with value for delay
L1          dec.w   R15                     ; decrement R15
            jnz     L1                      ; if R15 is not zero jump to L1
            jmp     Mainloop                ; jump to the Mainloop label

;-------------------------------------------------------------------------------
;   Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  _main

            .end
```

You may (or may not) wish to load and assemble the code prior to studying it. The following steps for creating an assembly project using CCS have been extremely abbreviated from the CCS User's Guide.

- From within CCS select File --> New --> CCS Project
- Enter the name for the project, click next and set Project Type to MSP430
- Click next twice to get to the CCS Project Settings page. Select the appropriate device variant, check "Configure as an assembly only project" and click Finish
- Select File --> New --> Source File
- Enter file name with the suffix .asm.
- Type or paste program text into the file.
- When code is complete select Project --> Build Active Project
- Select Target --> Debug Active Project
- Select Target --> Run to start the application.

First Some Housekeeping

```
            .cdecls C,LIST,"msp430g2231.h"  ; cdecls tells assembler to allow
                                            ; the c header file

;-------------------------------------------------------------------------------
;   Main Code
;-------------------------------------------------------------------------------

            .text                           ; program start
            .global _main                 ; define entry point

_main       mov.w   #0280h,SP               ; initialize stack pointer
            mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; stop watchdog timer
```

The first lines of our code repeated above are what might be called housekeeping tasks. They are necessary basic setups that will appear in one form or another in most of your programs. Unavoidably these setup tasks spill into a few concepts such as the "entry point" and the "stack pointer" that stray beyond the introductory notions intended for the tutorial. As these steps are examined these functions will be described, but most of the focus will be placed on the assembly language and syntax. When you set out to develop your own programs, you can at first simply copy these statements as a template.

Interspersed with comments, the first three statements are assembler directives. The .cdecls directive tells the assembler to allow the use of the C program header file. When you installed CCS it came with header files for all of the MSP430 variants. These files provide lots of useful information about the specific device you are programming, including the definitions for the symbols we will use to reference registers. The appropriate header file for the LaunchPad's MSP430G2231 is specified in quotation marks - msp430g2231.h. The .text directive indicates the beginning of a block of code. The .global directive is used here to declare the symbol _main as available across external modules so that when it is used in the next statement the assembler will recognize it as the entry point to the program.

The last two lines of this housekeeping code bring the first machine instruction of the program - mov. Here we use the mov instruction twice; once to set the initial value of the stack pointer and again to disable the watchdog timer. The stack is a special and important area of memory used to preserve values during subroutines, interrupts and when implementing certain machine instructions. The stack must be set at the beginning of microcontroller operation - typically at the top of memory. The watchdog timer is used to automatically restart the processor if a software problem occurs. The watchdog timer is active by default, so if it is not used in your program it should be disabled to avoid unexpected restarts.

The mov instruction loads the value indicated by the first operand (the source operand) into the location indicated by the second operand (the destination operand). Due to the robust and flexible nature of MSP430 addressing, the source operand might be a literal value to be loaded to the destination, or it might just as easily refer to a value stored in a register or other memory location. In the case of both of these lines, the source is the value of the operand. This addressing mode is called the immediate mode and it is indicated by the hash (#) preceding the operand. The statement:

```
            mov.w   #0280h,SP
```

loads the stack pointer with the number 280h, with the h indicating that this number is in hexadecimal format. The .w following the mov statement indicates that this is a word operation. All single-operand and dual-operand instructions can be byte (8-bit) or word (16-bit) instructions by using .b or .w extensions. If no extension is used the instruction is a word instruction, but there is no harm in including the .w for clarity. Because SP is one of the 16 registers that is part of the CPU itself, it uses register mode addressing and can be referenced here with no prefix.

The next statement:

```
            mov.w   #WDTPW+WDTHOLD,&WDTCTL
```

is more confusing at first glance. What we are doing with this mov instruction is loading the watchdog control register with a value that will disable the timer. WDTCTL is a symbolic alias equal to the address of the watchdog timer control register; it was defined in the header file. The ampersand (&) indicates that this operand will use absolute addressing mode meaning that the number represented by the alias is the address of the register. WDTPW and WDTHOLD are two aliases that were also loaded with the header. WDTPW is the value of the watchdog timer password which must be the first half of any operation that writes to the WDTCTL register. WDTHOLD is equal to bit 7 which is the stop bit of WDTCTL. When added together and loaded to WDTCTL they will stop the watchdog timer.

Setting Up the Inputs and Outputs

With the housekeeping tasks out of the way, it's time to start writing some code. The first thing we need to do is configure the microcontroller pins for input and output (I/O). The MSP430G2231 has 10 I/O pins each of which can be individually configured. Eight pins are implemented on port P1, and the remaining two pins are on port P2. Pins on a port are numbered 0 to 7 and referenced by the combination of their port and pin numbers such that Port 1 Pin 3 is P1.3.

Ports P1 and P2 each have a register - P1DIR and P2DIR respectively - that is used to control whether a pin is an input or an output. These are 8 bit registers with each bit corresponding to an I/O pin. If the bit is set then the pin is an output, if the bit is clear then the pin is an input. As with the pin numbers, bit numbers in a register always begin with 0; 8 bit registers are 0-7 and 16 bit registers are 0-15.

You will often need to speficy a register bit in the operand of a machine instruction. When expressing registers as numeric values, a binary number is created where each bit in the register corresponds to a digit in the number with the lowest bit (0) furthest to the right: bit 0 is expressed as binary 00000001 while bit 7 is expressed as binary 10000000. Note here that while the binary number represents a bit position, the actual value of the binary number is not equal to the decimal number of the bit. Binary 10000000 is equal to decimal 128 and hexidecimal 80; the assembler will accept any of those formats for bit 7 provided they are expressed in the proper syntax. You will frequently see hexadecimal used with these types of commands, but this is a matter of choice and convenience. Binary expressions can be helpful in the visualization of individual bits, but they are prone to errors that can be hard to find - especially when dealing with 16 bit numbers.

For this project we will be concerned with three of the I/O pins. On the LaunchPad, the red LED is connected to P1.0, the green LED is connected to P1.6, and the push button that we want to use is connected to P1.3. Therefore, in our example program the bits 0 and 6 of the P1DIR register will need to be set, while the P.3 will need to be clear. We could do this by once again using mov to load a value into the P1DIR register, however there is another instruction, bis - the "bit set" instruction, which is useful for setting individual bits and more frequently used for writing to the PxDIR registers:

```
            bis.b   #01000001b,&P1DIR       ; make P1.0 and P1.6 output
                                            ; all others are inputs by default
```

bis behaves quite differently than mov. When using bis, any bit that is a "1" in the source operand will be set in the destination operand, but the remaining bits will be unaffected. In this case the entire P1DIR register would have been clear on startup, so there is no need to do anything to clear the bit for P.3 and the pin configuration is complete.

Quickly reviewing some of the syntax used in this last statement, the .b extension to the mnemonic indicates this is a byte operation, the # prefix to the source operand indicates the use of immediate addressing, the b suffix of the source operand indicates a binary number, and the & prefix to the destination operand indicates absolute addressing.

The Main Loop

The user defined label Mainloop is a good indication that we have reached the heart of the program. As a label, Mainloop doesn't perform any action, but it serves as the marker for the start of a block of code that will repeat indefinitely as the program runs. Later in the code an instruction will be executed to branch program flow back to this point.

Reading the Button and Taking Action

These first two lines of the Mainloop code will work together to get the status of the LaunchPad's push button and use that status to determine whether or not to branch to a different section:

```
Mainloop    bit.b   #00001000b,&P1IN        ; read switch at P1.3
            jc      Off                     ; if P1.3 open branch to Off label
```

Here we are looking at a different register for P1, the P1IN register, which reflects the value of the input signals at the port's corresponding I/O pins. The bit or "bit test" instruction reads bits in the destination operand as indicated in the source operand - here bit 3. However, unlike the previous instructions mov and bis, the bit instruction does not change the destination operand, but instead only affects a group of bits of the CPU's status register conveniently called the status bits. In this case we will be concerned with the C or "carry" bit which is set when a bit test operation returns a 1. The jc or "jump if carry" instruction will read the carry bit and if it is set will branch to the location indicated by its only operand - here the label Off. If the carry bit is clear, program execution will continue with the next statement. MSP430 assembly language has seven different conditional jump instructions that are useful in creating such IF/THEN constructs.

As an important aside, note that the LaunchPad push buttons are wired so that the P1IN bit is set to a "1" when they are open and clear to a "0" when they are pushed. This is a common engineering practice but makes the code just a little counterintuitive at this juncture where we might expect a 1 to equal the on state.

If The Button is On

If the button is pushed and bit 3 of P1IN returns a 0, then the jc instruction will not jump to the Off label and execution will continue with the next block of code:

```
On          bic.b   #00000001b,&P1OUT       ; clear P1.0 (red off)
            bis.b   #01000000b,&P1OUT       ; set P1.6 (green on)
            jmp     Wait                    ; branch to a delay routine
```

The On label here is superfluous because nothing else in the code refers to it, but it makes this block a little more readable and does not add any overhead to program execution. The next two instructions will write to the P1OUT register, which as you might intuit by now is the register to control the state of P1 pins that have been set as outputs. The bic or "bit clear" instruction is the functional opposite of bis - any bit that is a "1" in the source operand is cleared in the destination operand, but the remaining bits will be unaffected. Together these lines clear bit 0 of P1 and set bit 6 which will ensure the red light is off and the green light is on. The jmp or "jump" instruction is an unconditional jump that will relocate program execution to the label Wait several statements further along in the code, skipping over the block which is intended to execute when the button is in the off condition.

If The Button is Off

If the button is not pushed and bit 3 of P1IN returns a 1, then the jc instruction branches execution to the Off label:

```
Off         bis.b   #00000001b,&P1OUT       ; set P1.0 (red on)
            bic.b   #01000000b,&P1OUT       ; clear P1.6 (green off)
```

This block of code is essentially a mirror image of the On block. The bis instruction turns on the red LED and the bic instruction turns off the green LED.

The Delay Loop

Regardless of whether the On or the Off blocks of code are executed, both paths lead to the next block of code which begins with the user defined label Wait:

```
Wait        mov.w   #1834,R15               ; load R15 with value for delay
L1          dec.w   R15                     ; decrement R15
            jnz     L1                      ; if R15 is not zero jump to L1
            jmp     Mainloop                ; jump to the Mainloop label
```

The wait routine is included to provide a rudimentary switch debounce. Mechanical switches are notoriously noisy when opening and closing, and they tend to "bounce" back and forth between states briefly before settling down. While the possibility of bounce doesn't present much of a problem in this simple application, we will address it for good measure and to introduce a very common assembly routine - the delay loop. As written here this loop will provide a delay of about 5 milliseconds (thousandths of a second) between readings as a simple way to give the switch a chance to stabilize.

First let's examine the code itself. The premise of the delay loop is to waste time by executing a non-functional routine through hundreds or thousands of iterations. The first line of this block uses the mov instruction to load the decimal value 1834 to R15. The MSP430 has 12 general purpose registers R4-R15 that are useful for the temporary storage of a value in situations just like this one. This value will be the number of times that the loop will iterate. The next two instructions form a loop that will subtract 1 from R15 on each iteration, effectively counting down until R15 equals zero. The dec or "decrement" instruction subtracts 1 from the destination operand and stores the result back to the operand. Additionally, the dec instruction will affect the Z or "zero" status bit. The Z bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0. The jnz or "jump if not zero" instruction reads the Z bit and if it is not set relocates program execution to the operand - here the label L1. When R15 finally equals 0, instead of looping to L1 program flow moves forward again to the next instruction where an unconditional jump sends execution all the way back to the Mainloop label and the whole process of reading the button and setting the LEDs will begin again.

The number of iterations for a delay loop is determined by calculating how many instruction cycles are in the loop and how many instruction cycles it will take to expend the required time for the delay. In its default startup mode the MSP430G2231 will run at about 1.1Mhz which gives an instruction cycle time of 909 nanoseconds (billionths of a second). Not every instruction will execute in one cycle. The number of cycles depends on the addressing mode and the particular instruction. Here the dec instruction takes 1 clock cycle and jnz takes 2 clock cycles, therefore the total time per iteration of this loop is 3 clock cycles. A basic formula would be Loops = Delay / (Cycle Time * Instruction Cycles in Loop) which in this case would be 1834 = 0.005sec / (0.000000909sec * 3).

This simple example is a starting point for understanding and working with delays and timing issues. Be aware that the MSP430 can be configured to operate at both higher and lower clock speeds and provides many more sophisticated options for extremely precise timing.

More Housekeeping

```
;-------------------------------------------------------------------------------
```

```
;   Interrupt Vectors
;------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  _main

            .end
```

The last lines of the program could be considered more housekeeping and again could simply be copied over as a template when you first write your own code. What is happening here is the reset vector is being configured to point back to the entry point of the program if a reset condition such as a brownout occurs. The .sect assembler directive creates named sections with relocatable addresses that can contain code or data. The .short directive initializes one or more 16-bit integers.

Finally the .end directive is optional and marks the end of the program. When used it must be the last source statement of a program as any subsequent statements will be ignored.

In Conclusion

If you have not already downloaded the following documents from the Texas Instruments web site, that should probably be your next step.

- Assembly Language Tools MSP430 User Guide (slau131e)
- MSP430x2xx Family User's Guide (slau144e)
- MSP430G2x21, MSP430G2x31 Datasheet (slas694b)
- Code Composer Studio Users Guide (slau157n)

All told these manuals contain over 1000 pages of critical information about the assembler tools, the instruction set, the electrical characteristics and features of the microcontroller family. One of the goals of this tutorial has been to present the MSP430 assembly language in as direct and simple a fashion as possible and provide a quick start to the programming experience. Nonetheless, it should be clear by now that learning to program the MSP430 is no trivial matter. Numerous significant topics were only briefly introduced and countless others did not even receive a mention. Hopefully, we've whet your appetite for this platform and established a foundation from which you can begin to build your understanding and toolset.

http://robotics.hobbizine.com/asmlau.html