Lab 2 – Part 3
Assembly Language Programming and 9S12 Ports

Introduction and Objectives

In this week's lab you will write an assembly language program to display various patterns on the eight individual LEDs on your Dragon12-Plus board. The displayed pattern will be based on the state of two bits of the onboard DIP switch. You will also start using subroutines, and investigate the stack and stack pointer, and learn how to load your program into EEPROM so the program will remain on your board after a power cycle.

The program for this lab will display four different patterns on the LED display connected to Port B. You will use the state of bits 1 and 0 of the onboard DIP switch to select which of the four patterns to display.

1. Prelab

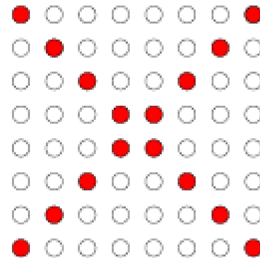
Write a program to set up Port B as an eight bit output port (be sure to disable the seven-segment displays, and to enable the individual LEDS), as you did in last week's lab, and to implement (i) a binary up counter, (ii) a shifting bit, (iii) a Johnson counter, and (iv) a Ford Thunderbird style turn signal based on the state of the DIP switches. Insert a 100 ms delay between updates of the display. Write the delay as a subroutine. Be sure to initialize the stack pointer in you program.

(Note: you will be referring to a number of MC9S12 registers in this and future programs. It is tedious and error-prone to look up and enter the addresses of the registers each time you write a new program. There is a file when you start a new project called 'directive.inc' that links to a file which has a list of all registers and their addresses for the 9S12DP256 version of the MC9S12 microcontroller. If you include that file in your program (by including the line INCLUDE 'directive.inc' as the first line of your program), you can refer to all registers by name rather than having to look up their addresses.)

1. Write a subroutine to implement the binary up counter, have the LEDs count 0, 1, 2, 3, 4, ... It should take 256 counts from the time all LEDs are off until the next time all are off again.

2. Write a subroutine to display a shifting bit on PORTB which looks like the figure below. There is an easy way to calculate this. Start with two variables, one with a value of 0x80 and the other with a value of 0x01. OR the two variables together to get 0x81,

the first pattern in the sequence. Then rotate the first variable right by one (to get 0x40), and rotate the second variable left by one (to get 0x02). OR these two together to get 0x42, the second pattern in the sequence. Continue rotating the first variable to the right, the second to the left, and ORing the two together.



3. Write a subroutine to generate the next pattern in the sequence for an eight-bit Johnson counter. The procedure to do this is as follows: Shift the present pattern to the right by one bit. The most significant bit of the next pattern is the inverse of the least significant bit of the present pattern. The number to convert is in accumulator A, and the next pattern in the sequence is returned in accumulator A. The subroutine should return with all registers, except A, the same as when the subroutine was called, so use the stack to save and restore any registers you need to use to implement the subroutine. The starting pattern is 00000000;

4. Write a subroutine to take the next entry out of a table, write it to PORTB, and update the index into the table. Here is an example of what the table might look like:

```
table_len:    equ (table_end-table)
              org data
table:        dc.b $00, $01, $02, $04, $08, $10, $20, $40, $80
table_end:
```

The index of the number to be displayed is passed in accumulator A. Your code should write the table entry corresponding to that index to PORTB. Return the index to the next table element in accumulator A. (For example, if accumulator A were 5, you would write the fifth element of the table, $10, to PORTB, and return a 6.) Make sure that the index stays between 0 and table_len - 1. The subroutine should return with all registers, except A, the same as when the subroutine was called, so use the stack to save and restore any registers you need to use to implement the subroutine. Fill out the table such that you get the following pattern. (This is called the TBird Taillights, because the old Ford Thunderbirds displayed this type of pattern for their turn signals. Current TBirds no longer display this type of pattern.)

5. Write the program that will display four different patterns on the LED display connected to Port B. You will use the state of bits 1 and 0 of the onboard DIP switch to select which of the four patterns to display. Write a program to set up Port B as an eight bit output port (be sure to disable the seven-segment displays, and to enable the individual LEDs), and to implement (i) a binary up counter, (ii) pattern from part 2, (iii) a Johnson counter, and (iv) a Ford Thunderbird style turn signal based on the state of the DIP switches. Insert a 100 ms delay between updates of the display. Write the delay as a subroutine. Be sure to initialize the stack pointer in you program. Use four variables to hold information on the four patterns. Initialize these four variables to the first pattern in the sequence. You should have a loop which checks the DIP switches connected to Port H. If bit 7 of the DIP switches is high, end the loop and exit back to DBug-12 with a SWI instruction. If bit 7 of the DIP switches is low, check bits 0 and 1 to determine what

PH1	PH0	Pattern
0	0	Binary Up Counter
0	1	Pattern from Part 2
1	0	Johnson Counter
1	1	TBird Turn Signal

For example, if bits 1 and 0 of Port H are 10, load accumulator A with the Johnson Counter variable, call the Johnson Counter subroutine, and save the returned accumulator A into the Johnson Counter variable. Call the Delay subroutine, then loop back to check the DIP switches again.

2. The Lab

1. Implement the program described in the prelab. If you have difficulty getting your program to work, start by trying to implement one function only say, the binary counter. Once this works, start working on your next functions. Verify that all the functions work correctly.

2. The MC9S12 has EEPROM (Electrically Erasable Programmable Read Only Memory) functionality. If you put your program into EEPROM the program will remain there when you turn off power.

(a) The EEPROM is located at address 0x400. You can just change the origin statement of your assembly language program, then reassemble, and reload your program. (Loading programs into EEPROM takes a longer time than loading programs into RAM. DBug-12 needs to tell the Hyberterminal to wait while it programs some EEPROM bytes before it sends the next set of bytes to program. It uses a protocol called Xon/Xoff to do this. Make sure Hyperterminal is set up to use Xon/Xoff. Use the MD command to verify that your program was correctly loaded into EEPROM. Type G 400 to run your program out of EEPROM. It should work the same as it did when you ran it out of RAM. (Try it.)

You can power cycle your board, and then type G 400, and again your program will run correctly. (Try it. The TBird pattern may not work correctly. The reason for this and the solution is discussed below.)

For some applications it would be nice if you could run your program without having to type G 400 if your board is controlling a robot, and no computer is connected to it, it would be impossible to start the program by typing G 400. DBug12 has a special mode to allow you to run a program out of EEPROM without having to type G 400. If you set the two switches on the LOAD DIP switch to Jump to EEPROM mode (Switch 2 on, Switch 1 off), and power cycle the board (or push the reset button), the program will run immediately out of EEPROM. (Try it.)

You will notice that the program runs much slower actually, six times slower than it did when you ran it by typing G 400. This is because DBug12 does some system initialization which is bypassed when you run your program directly from EEPROM. In particular, the Dragon12-Plus board has an 8 MHz clock, and the MC9S12 runs at half the clock frequency, or 4 MHz. The MC9S12 has a built in phases lock loop (PLL) which allows the chip to generate a faster clock internally, and run with a 24 MHz E-clock frequency. In order to get the chip to run at the higher frequency, you must do the initialization which enables the PLL. Here is some code which will do that initialization (adapted from the Dragon12-Plus Reference Manual):

```
; PLL code for 24MHz bus speed from an 8 crystal
            sei                         ; disable interrupts
            bclr CLKSEL,%10000000       ; clear bit 7, clk derived from oscclk
            bset PLLCTL,%01000000       ; Turn PLL on, bit 6=1 PLL on, =0 off
            movb #$05,SYNR              ; 5+1=6 multiplier
            movb #$01,REFDV             ; divisor=1+1=2,8*2*6/2=48MHz PLL
```

```
                                                    ; freq, for8MHz crystal
wait_b3      brclr CRGFLG,%00001000,wait_b3 ; wait until bit3=1
             bset CLKSEL,%10000000     ; derive clock from PLL
```

(b) Add the above code to your program, right after the org $400 line and before the first line of your program. Load this new code into EEPROM. (Be sure to move SW1 of the LOAD DIP switch down in order to get back to the DBug12 monitor so you can load new code into memory.) Now move SW1 of the LOAD DIP SWITCH to the up position, power cycle your board, and your program should run at the same speed it did when running out of RAM.

(c) Another problem with running out of EEPROM is that data which is loaded when you load your program is not present when you start your program out of EEPROM after a power cycle. For example, if the TBird pattern is put into RAM, when you turn power off that pattern is lost, and when you turn the power back on and start running the program from EEPROM, an incorrect pattern is displayed. To fix this, put the table into the program section of memory rather than the data section. In this way, the table is programmed into EEPROM as well your program. Now if you power cycle the board, the table with the TBird pattern is still there. When you put a program into EEPROM, only variables which change should be put into the data section. Also, you need to initialize these variables in the program rather than using a dc.b directive.

Note: The document readme EEPROM.pdf which came on the DRAGON12-Plus CD says that you need to convert your S1 code (in the S19 file) to S2 code to successfully load a program into EEPROM. This is because the MC9S12 EEPROM must be programmed with an even number of bytes, and must be programmed starting at an even address. However, I have had no problem loading a program which starts on an odd address or has an odd number of bytes. I think that, when DBug12 sees that a user wants to load a program which starts on an odd address or contains an odd number of bytes, into EEPROM, it automatically adds the bytes needed to make the program start on an even address or to contain an even number of bytes. If you have trouble getting an EEPROM program to load correctly, you should try converting your S1 code to S2 code as discussed in the readme EEPROM.pdf document.