FINAL PROJECT: INTERFACING AND MOTOR CONTROL

In this sequence of labs you will learn how to interface with additional hardware and implement a motor speed control system.

WEEK 1

PORT EXPANSION FOR THE MC9S12

Introduction and Objectives

It is sometimes necessary to add additional memory and/or hardware to a microprocessor or microcontroller. Interfaces such as the SPI allow you to add some hardware, it is often necessary to interface directly to the address/data bus. For a microprocessor, which does not have built-in peripherals, the address/data bus is the only way to add additional memory or hardware. In this lab you will add an extra output port to your MC9S12.

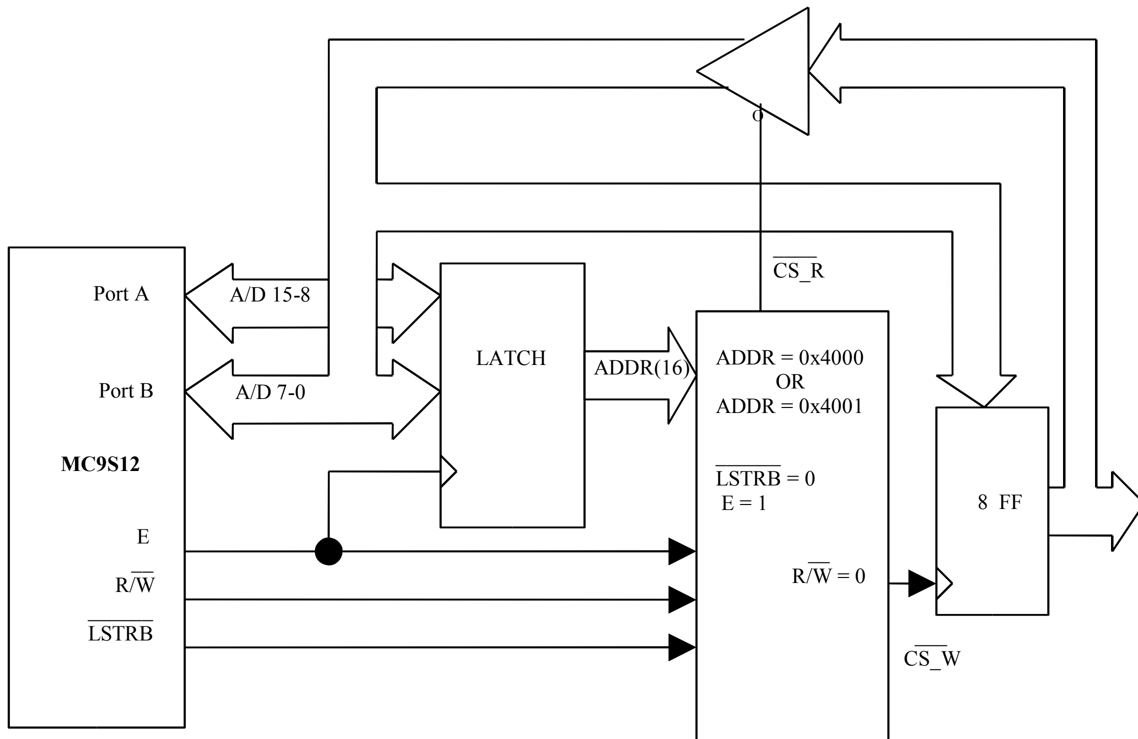

Figure 1. Block diagram of HCS12 output port at address 0x4001

Writing to address 0x4001 will bring CS_W low.
On the high-to-low transition of E, CS_W will go high, latching the data into the flip-flops

Reading from address 0x4001 will bring CS_R low
This will drive the data from the flip-flop onto the data bus
The MC9S12 will read the data on the flip-flops on the high-to-low transition of the E-clock.

Figure 1 shows a block diagram for adding an external output port to the microcontroller. We will implement the port in an Altera 7064 PLD. Note that you will have to connect the 16-bit multiplexed address/data bus and three control lines from your MC9S12 to your Altera chip. You will also have to connect the eight bits of your output port to your LEDs to verify that the port is working.

1. Write an Altera program to generate CSW and CSR. Add this code to the Altera program which will be supplied in lab.

2. Using the pinout diagram from the .rpt file, wire the Altera chip to your MC9S12. Note that there will be a lot of wires to run, so it is essential that you are neat in your wiring.

3. Check the functioning of your port using D-Bug12. When you start your MC9S12 using D-Bug12, the microcontroller is in single chip mode. In this mode you can manipulate AD-15-0, E, R/W and LSTRB as general purpose I/O lines. You can use the MM command of D-Bug12 to write data to the output port by changing AD15-0 (PORTA and PORTB), E, R/W and LSTRB in the same sequence that the MC9S12 would if it were in expanded wide mode. Look at Chapter 12 of the HCS12 Core Users Guide for more information.

   a) Use the DDRE register to make E, R/W and LSTRB output pins. (Note: E is bit 4 of PORTE, R/W is bit 2 of PORTE, and LSTRB is bit 3 of PORTE.
   b) Bring E low by writing to PORTE.
   c) Put 0x4001 on PORTA and PORTB.
   d) Bring R/W and LSTRB low.
   e) Bring E high.
   f) Put the data you want to write to the port on PORTB.
   g) Bring E low.

4. Now you will switch the MC9S12 into expanded wide mode, so the MC9S12 can write directly to the expanded output port using its address/data bus. Unfortunately, D-Bug12 does not work when the chip is put into expanded wide mode. To verify that the expanded port works, you will have to put your program into EEPROM, along with the instructions needed to switch the MC9S12 into expanded wide mode. Then by having the MC9S12 execute EEPROM code after reset, you will be able to see that the expanded port works properly. Also, our MC9S12 has no region of free memory in which to map our new port. We can use the MISC register to turn off the Flash EEPROM from 0x4000 to 0x7fff, which will give us a region in which to put our port. The following program will do all this:

```
PEAR:       equ $000a
MODE:       equ $000b
EBICTL:     equ $000e
INITRM:     equ $0010
MISC:       equ $0013
SYNR:       equ $0034
REFDV:      equ $0035
CRGFLG:     equ $0037
CLKSEL:     equ $0039
COPCTL:     equ $003c
ARMCOP:     equ $003f

            org $400

            lda     #$55            ;reset COP
```

```
            staa    ARMCOP
            coma
            staa    ARMCOP
            clr     COPCTL              ; turn off COP
            ldab    #$11                ; map RAM into proper location
            nop
            stab    INITRM
            ldab #$00                   ; set clock reference divider to 0
            stab REFDV
            ldab #$03                   ; set PLL to multiply oscillator clock by 4
            stab SYNR
            nop                         ; wait
            nop
            nop
            nop
l1:         brclr CRGFLG,#$08,l1        ; wait for PLL to lock
            bset CLKSEL,#$80            ; switch to PLL clock
            ldab #$e8                   ; expanded wide mode, internal visibility on
            stab MODE
            ldab #$0c                   ; turn on R/W, LSTRB
            stab PEAR
            ldab #$01                   ; use E-clock to control external bus
            stab EBICTL
            stab #$03                   ; no E-clock stretch, disable ROM form 4000-7fff
            stab MISC
```

Add to the above program code which will increment the external port, with a delay between incrementing. Implement the delay using the RTI interrupt. Because you will not be using Dbug-12, you cannot use it to load your interrupt vector. You will have to write the address of the rti_isr interrupt routine to the interrupt vector in your program, something like this:

```
RTI_VEC:    equ $3e70

            ldx #rti_isr
            stx RTI_VEC
```

You should use the command

```
            inc $4001
```

to increment the value on the expanded port.

5. An MC9S12 with a functioning expansion port will be available at one of the logic analyzers during lab this week. The MC9S12 is running the following loop:

```
            org $0480

loop:       ldx $4000
            inc $4001
            ldaa $4000
            bra loop
```

The label **loop** is at address 0x0480.

a. Hand-assemble this program to determine the op codes and op code addresses.

b. Use the logic analyzer to grab data from the MC9S12 address/data bus. Identify the memory cycle which reads data from address 0x4001, and the memory cycle which writes data to address

0x4001. Note that the logic analyzer has only 16 data lines. The MC9S12 address/data bus uses 19 lines – AD15-0 and the three control line E, R/W, and LSTRB.

The HCS12 will either be fetching instruction from EEPROM (address 0x0400-0x0fff), or accessing the external port at 0x4001. Thus, address bits D15, D13, D12 will always be zero. These three lines will not be connected to the logic analyzer.

Figure A-9 of the MC9S12DP256B Device Users Guide shows the external but timing. As best you can, measure the following times. The numbers in parentheses are the labeled numbers on Figure A-9 and Table A-20. Compare the numbers to the values listed in Table A-20.

i. Cycle time (2)
ii. Pulse width, E low (3)
iii. Pulse width, E high (4)
iv. Address delay time (5)
v. Muxed address hold time (7)
vi. Write data hold time (13)
vii. Read/Write delay time (24)
viii. Read/Write hold time (26)
ix. Low strobe delay time (27)
x. Low strobe hold time (29)

Pre-Lab

1. Write a preliminary Altera program to generate the CSW and CSR.

2. Hand assemble the instructions of Part 5.

WEEK 2

MOTOR SPEED CONTROL

Introduction and Objectives

In this lab you will control the speed of a motor. Figure 1 shows the hardware setup, which will be the same as for Week 1 of Lab 4. You will see a potentiometer on your evaluation board to set the desired speed of the motor, and you will control the speed through the PWM output of the HCS12. You will measure the speed of the motor using an input capture pin, and display the desired and actual speeds on the terminal.

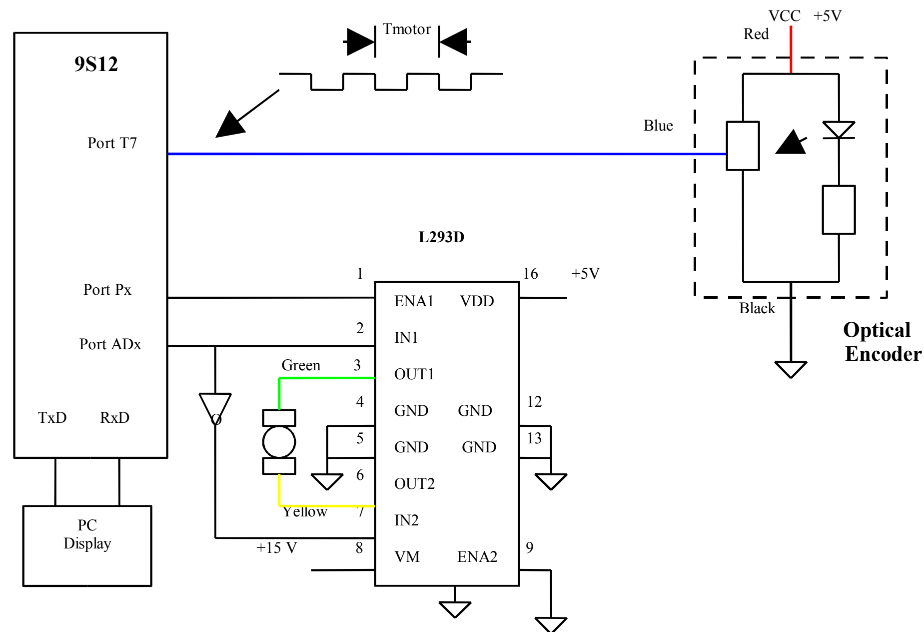

Figure 1. Block diagram of HCS12 output port at address 0x4055

1. Build the circuit shown in Figure 1.
2. Set up the RTI to generate an interrupt once very 8 ms. In the interrupt service routine, increment LEDs connected to Port A. Verify from the rate at which the LEDs are incrementing that you are getting interrupts at a rate of 8 ms.
3. Program the A/D converter to read the value from the pot. Use 8-bit A/D mode. In your RTI ISR, read the A/D converter, and write the eight most significant bits to Port A. In the main program loop, print the value read from the A/D to the terminal. (Do not print inside the ISR – this will take more than 8 ms, and you will miss interrupts.) Verify that the A/D values change as expected as you use the pot to change the voltage.
4. Set up the PWM to generate a 50 kHz PWM signal on one of the four PWM channels. Set it up for high polarity. It will be easiest to set PWPERx to 255.
   Verify that the PWM works. In the RTI ISR, write the eight most significant bits to the A/D value you read to PWDTYx. The motor speed should change as you sue the pot to vary the voltage on the A/D.

5. Measure the speed of the motor. Set up an Input Capture interrupt to determine the time between two falling edges of the optical encoder. In your main program write this time difference to the terminal.
6. Measure the speed for several different duty cycles by varying the voltage with the pot. Plot speed vs. duty cycle.
7. Implement closed-loop speed control. The desired speed S_d should be

$$S_d = (0.2 + 0.8 \frac{AD}{AD_{max}}) S_{max}$$

where Smax is the motor speed at 100% duty cycle, AD is the A/D converter reading, and ADmax is the maximum A/D converter reading. In this way you will be able to vary the speed between 20% and 100% of Smax.

To set the motor at the desired speed you can use a simple equation (proportional control) such as:

$$DC_{new} = DC_{old} + k(S_d - S_m)$$

where Sm is the measured speed. Do this calculation inside the RTI ISR, and write the new value to PWDTYx. Try different values of k to see how the motor responds. If k is too small, it will take a long time for the motor to get to its steady-state speed. If k is too large, the motor will be jerky as it tries to settle down to its steady-state speed.

It will be easier to do these calculations using floating point number rather than using integers. You can use floating point numbers with the GNU compiler. In the EmbeddedGNU IDE, select the Options menu, Project Options submenu. Near the bottom of the pop-up window, add the following to the Compiler options:

-fshort -double

By adoing this, you will be able to do basic operations with floating point numbers (add, subtract, multiply, divide). Do not try to use functions which require the math library (such as sqrt( )); the code generated by the GNU compiler will be too large to fit into the MC9S12. To print out a floating point number you must first convert it to an integer. For example,

```
float x;

x = 10.2;
DB12FNP -> printf("x=%d \r\n",(short)x);
```

If you use this method to print x when x is, say, 0.023, the value printed out will be zero. You could use the following to print a usable value for x:

```
DB12FNP -> printf("x=%d/1000 \r\n",(short)(x*1000.0));
```

The output from this when x is 0.023 will be 23/1000.

8. Measure the motor speed for various values of input voltage. Take about 10 equally-spaced measurements for input voltage between 0 and 5 V.

9. With the pot set at about mid-range, vary the voltage of the voltage powering the motor (say between 8 V and 14 V). With closed-loop control the speed of the motor should stay the same. Verify that this is the case.
10. Using the data from Part 8, plot the speed in RPM vs. the input voltage from the port – i.e., convert the speed measurement in time difference between two falling edges to speed in RPM.