EGR 222 Mechatronics
Lab #7 Micrococontroller Startup

March 1,2, 2011

7.0 Objectives:

Become familiar with the most basic principles of microcontroller operation and support tools. Specifically, students should be able to use the development support program "Code Warrior" to develop simple programs, and to download, run, and debug those programs for a Freescale HCS08QG8 microcontroller.

7.1 Pre-Lab assignment

Read Chapter 15 up through 15.3.1, 15.3.4, and at least skim Chapter 16, paying particular attention to the material covering particulars of the Motorola 68HC11, which is somewhat similar to our microcontroller. You should have the "DEMO9S08QG8" microcontroller demo board with its associated software and documents. You may wish to install "Code Warrior" and associated files on your own computer. We will have serial port connectors in the lab for the computers there, but if you don't have one and will want to use the serial port adapter eventually for your own computer, USB to serial adapters can be found still at stores carrying electronics. This is not really necessary for what we will be doing in this course. You may also want to visit the Freescale (formerly Motorola Semiconductor) web site for documentation on the HCS08QG8, or copy that documentation from the CD in the kit. There are some "Getting Started" documents that come with the kit: The "Quick Start Guide", and "User Guide" are helpful, and you could try out the board and experiment with using Code warrior ahead of class, which will be helpful.

7.2 Run the pre-installed test program described on the "Quick Start" document

Take out and examine the DEMO9S08QG8 board. Find the various important objects on the board, particularly the pushbuttons, LED's, and the USB connection for debugging support. See Figure 1.

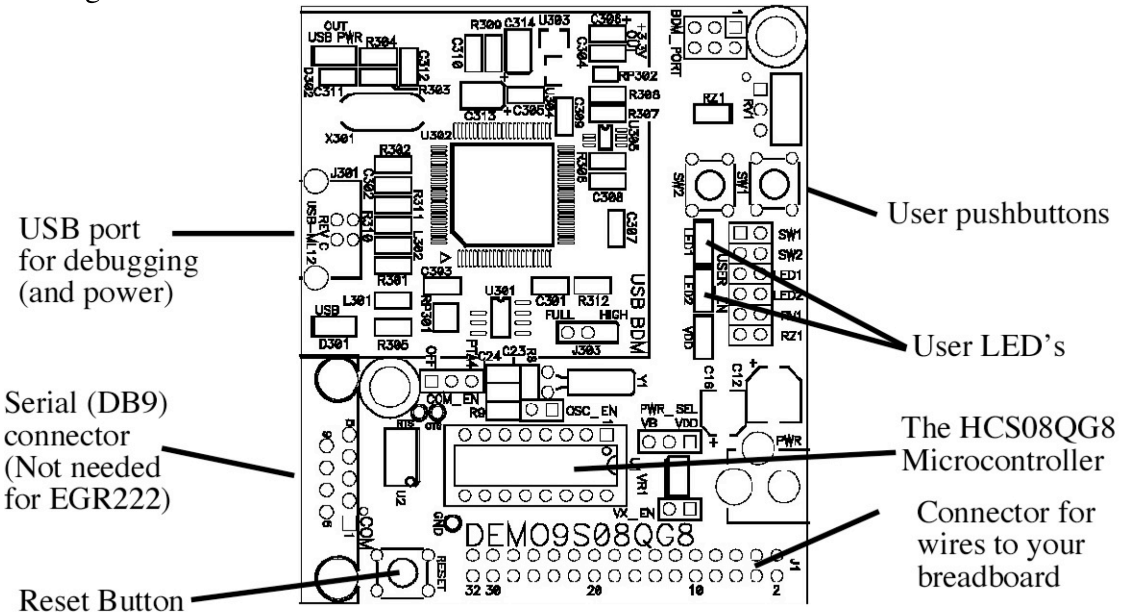

User pushbuttons
USB port for debugging (and power)
User LED's
Serial (DB9) connector (Not needed for EGR222)
The HCS08QG8 Microcontroller
Connector for wires to your breadboard
Reset Button
DEMO9S08QG8
USB BDM

Figure 1 DEMO9S08QG8 Microcontroller Demonstration Board

As the DEMO board comes, the “straps” for various signals should be in the correct places, as shown in Figure 2 below, but it’s not a bad idea to check them. Be sure to follow appropriate precautions to protect your board from static discharge. Touch the computer’s frame (ground) while touching ground of the board (the DB9 connector shell) to ensure that there is not a charge that will cause damage before plugging in the board. Then plug the board into a USB port on your computer using the supplied cable. When you do this the first time, the PC will have to look for the USB driver. It’s on the Code Warrior CD, which needs to be in your CD disk drive. Follow the directions of the installation wizard. For the lab computers, the driver should already be installed.

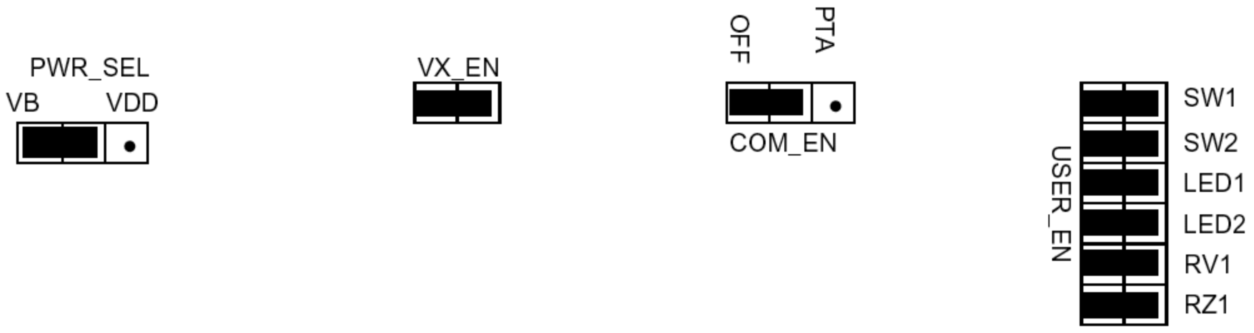


Figure 1 Strapping options for Demo Board

Once you plug in the board, it will be powered, and the simple installed program will begin running immediately. One of the LED’s will blink rapidly, and the other will toggle each time you press pushbutton “SW1”.

(You could try doing the exercise on the flip side of the “Quick Start” sheet, but I found that there was an error in the code that prevented it from compiling correctly that had to be fixed. You need to use the serial port, and Hyperterminal (or another terminal emulator) on the PC. It’s more complex (Much more complex) than the things we need to do in this lab exercise.)

7.3 Run Code Warrior and write code for a simple assembly language program

1) Start up “Code Warrior” from the Start menu. “Start using Code Warrior” to just start it.
2) Choose “New project” from the File menu. You will go through several “wizard” steps.
3) Choose the microcontroller. Successively choose “HCS08, HCS08Q family, and finally the MC9S08QG8 to identify the microcontroller you are using. Set “Connections” to P&E Multilink / Cyclone Pro.
4) Choose your language. At first, we will use “relocatable assembly”. Name your project. I suggest your initials xyz for “Lab7xyz1” or something like that. You can also designate where to put your files. You can put them on your thumb drive or H drive, or just to the desktop. If you do use the desktop, remember to copy your files to somewhere more permanent before you leave.
5) Do NOT add existing files to your project. (You might want to do that later for another project, but not this time. Usually you do want to copy when you add files.) Leave checked “create main.asm” so that the file for your main (and only) program will be created automatically.
6) Indicate “none” for Processor Expert. Later on you might want to revisit and play with this.

7) Now you have a project. On the left is a “view” of the project that shows what it contains. One of those is the main program code file, “main.asm”. Double click on it to open it up in the main window of “Code Warrior.” It should initially appear as shown in Figure 3.

```
;*******************************************************************
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more     *
;* advanced functionality of this processor, please see the       *
;* demonstration applications, located in the examples            *
;* subdirectory of the "Freescale CodeWarrior for HC08" program    *
;* directory.                                                      *
;*******************************************************************

; Include derivative-specific definitions
            INCLUDE 'derivative.inc'

; export symbols
            XDEF _Startup, main
            ; we export both '_Startup' and 'main' as symbols. Either can
            ; be referenced in the linker .prm file or from C/C++ later on

            XREF __SEG_END_SSTACK   ; symbol defined by the linker for the
end of the stack


; variable/data section
MY_ZEROPAGE: SECTION  SHORT         ; Insert here your data definition

; code section
MyCode:     SECTION
main:
_Startup:
            LDHX   #__SEG_END_SSTACK ; initialize the stack pointer
            TXS
            CLI                     ; enable interrupts

mainLoop:
            ; Insert your code here
            NOP

            feed_watchdog
            BRA    mainLoop
```

Figure 3 Initial “Empty” main Program

8) Now, what you want to do is modify the main program to do what you want. Our initial program will be to blink the two user LED’s at a rate that depends on the pushbuttons. So, edit the main.asm file as follows:

a) Add a variable “i” by reserving space for it in the “MY_ZEROPAGE section, by adding the statement:

```
i:          DS 2
```

The “i:” is a “label” that identifies something in your code, in this case two bytes of space you have reserved using the “DS” (Define Space) directive.

b) Delete the line of code that enables interrupts, the “CLI” instruction (Clear Interrupt).

c) Add code that will set up all of the pins of Port A to be inputs, including the pins 2 and 3 for the pushbuttons. Do that by adding, before the “main loop” the following four instructions:

```
            LDA #$00
            STA PTADD       ;all pins of port A to be inputs
            LDA #$1f
            STA PTAPE       ;enable pull-ups on Port A
```

First we put the value $00 (8 bits of zeros, expressed as a hexadecimal number) into the Accumulator (A). Then we store A into the address for the Port A Direction Register, known as “PTADD”. This register is actually at address $0002. Then we store hexadecimal $1f (the binary value 00011111) into the Port A Pullup Enable register (PTAPE) so that there are resistors pulling up these signals to “1” in the absence of a switch or pushbutton pulling them down. In the case of pins 2 and 3, that’s exactly what SW1 and SW2 will do. The symbols “PTADD” and “PTAPE” are defined in the file “derivative.inc” which in turn includes them from “an include file for this particular microcontroller.

d) Add code that sets up Port B so that the top two bits are outputs (to the LEDs).

```
            LDA #$c0
            STA PTBDD       ;Bottom pins of Port B to be inputs, top 2 outputs
            LDA #$3f
            STA PTBPE       ;enable pull-ups on Port B inputs
```

Note that we indicate outputs by putting a “1” in the data direction register, so bits 6 and 7, which are wired to the LEDs (via the straps), are now connected to outputs instead of inputs.

e) Now, within the “main loop” (which will be executed over and over again), put code to turn the LED’s on:

```
mainLoop:
            ; my main loop code here
            LDA PTBD        ;turn on LEDs
            AND #$3f
            STA PTBD
```

What we do is load the current value of Port B, zero out the top two bits (by doing a bitwise AND of the 8 bits that had been in Port B with binary 00111111), then we store that back into Port B. After doing so, the top two bits will be 0. This causes an output low Voltage on the corresponding pins, sinking current that flows through the LED’s, lighting them.

f) Now read in the two bits from the pushbuttons:

```
            LDA PTAD        ;load values of port A pins 2 and 3
            AND #$0c
```

We “mask” off the bits we are not interested in by doing a bitwise “AND” here too. This time we AND with the binary value 00001100 (expressed as hexadecimal $0c) so only bits 2 and 3 are left nonzero.

g) We want to convert that into a larger number, so we shift the value left three times and store it into the variable “i”. Since we store it in the top half of I and zero out the bottom half, we actually have now 5+8 zeros following the two bits that started in port B bits 2 and 3.

```
            LSLA            ;shift the values of pins 2,3 into bits 5,6
            LSLA
            LSLA
            STA i           ;store i = (pins2,3)<<total of 13 bits
            CLRA
            STA i+1
```
h) Now we add code that will count down the value of I, one at a time. The purpose is just to waste time. This is how we will wait until some time elapses before we turn the LEDs off. Note how we use labels to identify where to go if the condition is not zero.
```
Loop1:      DEC i+1         ;decrement i
            BNE Skip1
            DEC i
Skip1:      LDHX i          ;see if it is zero.  If not, decrement again
            BNE Loop1
```
Note that here we use some "conditional" statements, "Branch if Not Equal to zero (BNE). DEC decrements first the lower 8 bits of I, and if that is zero we decrement the top part. Not exactly counting correctly but we do spend a lot of time doing it.

i) Now we need code to turn the LED's off:
```
            LDA PTBD        ;turn off LEDs
            ORA #$c0
            STA PTBD
```
j) That's followed by statements to again read the switches and count down, just as we did above.
```
            LDA PTAD        ;load values of port A pins 2 and 3
            AND #$0c
            LSLA            ;shift the values of pins 2,3 into bits 5,6
            LSLA
            LSLA
            STA i           ;store i = (pins2,3)<<total of 13 bits
            CLRA
            STA i+1
Loop2:      DEC i+1         ;decrement i
            BNE Skip2
            DEC i
Skip2:      LDHX i          ;see if it is zero.  If not, decrement again
            BNE Loop2
```
j) Delete the "feed the watchdog" statement (you can add a semicolon to comment it out.)
```
           ; delete feed_watchdog
            BRA mainLoop    ; go back to the beginning of the main loop
```

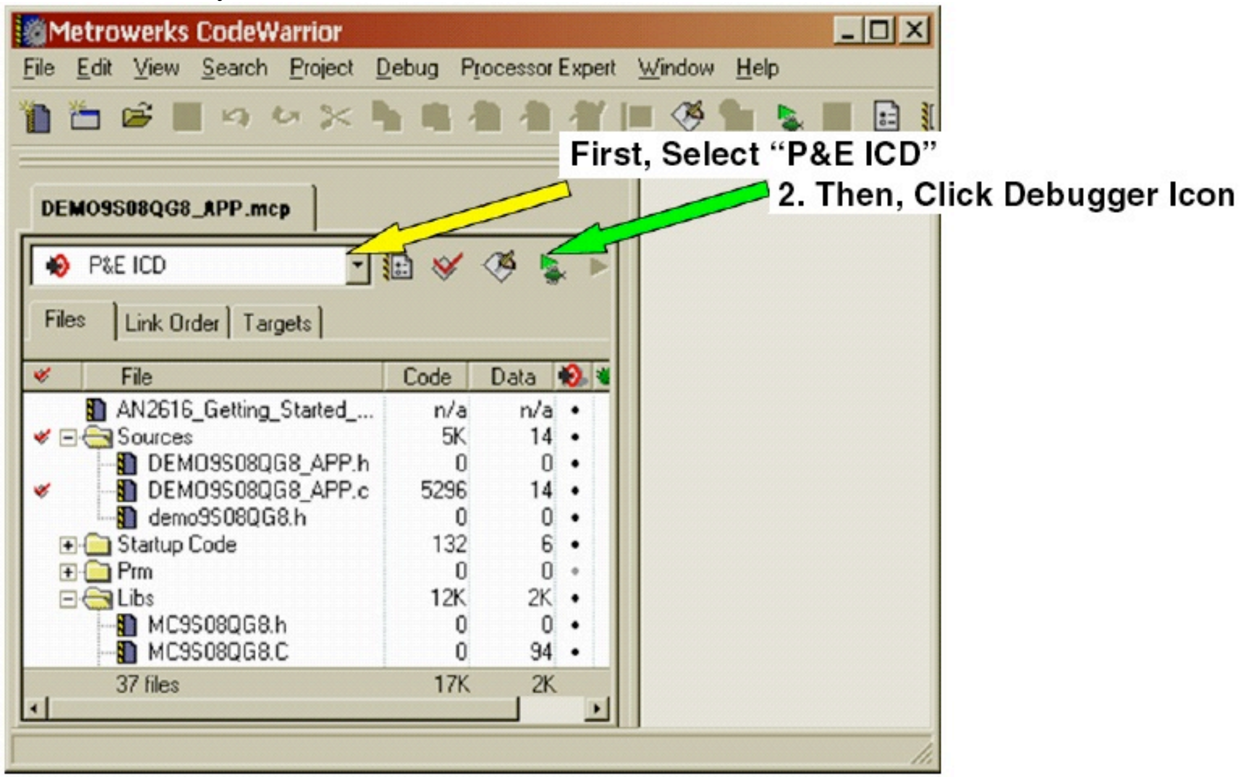
Now you have your main program. Save it. As completed, it should look something like that shown in Figure 4. Below:

```
;*******************************************************************
;* Application to blink the LED's at rate determined by switches   *
;* JBG on January 14, 2011, Wilkes University                      *
;*******************************************************************
; Include derivative-specific definitions
            INCLUDE 'derivative.inc'
; export symbols
            XDEF _Startup, main
            XREF __SEG_END_SSTACK; symbol defined by the linker for the end of stack
; variable/data section
MY_ZEROPAGE: SECTION  SHORT         ; Insert here your data definition
```

```
i:          DS 2
; code section
MyCode:     SECTION
main:
_Startup:
            LDHX   #__SEG_END_SSTACK ; initialize the stack pointer
            TXS
            ; delete the CLI, start my code here.
            LDA #$00
            STA PTADD      ;all pins of port A to be inputs
            LDA #$1f
            STA PTAPE      ;enable pull-ups on Port A
            LDA #$c0
            STA PTBDD      ;Bottom pins of Port B to be inputs, top 2 outputs
            LDA #$f
            STA PTBPE      ;enable pull-ups on Port B inputs
mainLoop:
            ; my main loop code here
            LDA PTBD       ;turn on LEDs
            AND #$3f
            STA PTBD
            LDA PTAD       ;load values of port A pins 2 and 3
            AND #$0c
            LSLA           ;shift the values of pins 2,3 into bits 5,6
            LSLA
            LSLA
            STA i          ;store i = (pins2,3)<<total of 13 bits
            CLRA
            STA i+1
Loop1:      DEC i+1        ;decrement i
            BNE Skip1
            DEC i
Skip1:      LDHX i         ;see if it is zero.  If not, decrement again
            BNE Loop1
            LDA PTBD       ;turn off LEDs
            ORA #$c0
            STA PTBD
            LDA PTAD       ;load values of port A pins 2 and 3
            AND #$0c
            LSLA           ;shift the values of pins 2,3 into bits 5,6
            LSLA
            LSLA
            STA i          ;store i = (pins2,3)<<total of 13 bits
            CLRA
            STA i+1
Loop2:      DEC i+1        ;decrement i
            BNE Skip2
            DEC i
Skip2:      LDHX i         ;see if it is zero.  If not, decrement again
            BNE Loop2
            ; delete feed_watchdog
            BRA mainLoop   ; go back to the beginning of the main loop
```
Figure 4 Example code

7.4 Assemble, Download, and Run your example program

Save your main program file. Either just above the project view, or in the menu bar at the top, there is a little green arrow pointing left for “Debug” and a symbol like a diamond just to it’s left to “make” the project as shown in Figure 5 below. Notice in the project view that there is a red checkmark beside the “main.asm” file indicating it has not yet been assembled. (There are checks for other components too.) The “make” the project, click the “make” symbol. This causes the “assembler” to convert each file into machine language, binary ones and zeros (usually written in hexadecimal). Then the linker connects such files into an “executable” image to be downloaded into the microcontroller. When you click “Debug”, another application is started up, the debugger, which begins by establishing a connection to the microcontroller (using the debug hardware on the board via USB). Note that in Figure 5, setting the communications is also shown, but that shouldn’t be necessary since you designated communications for debugging in project set-up. It will then ask if you want to erase the existing code in the microcontroller and replace it with what you have just “made.” (Yes, you do; take the default.) The debugger will then go through a lengthy process of copying your application into the microcontroller. At the end, it is ready to run.



	File	Code	Data	
	AN2616_Getting_Started_...	n/a	n/a	•
✓	Sources	5K	14	•
	DEMO9S08QG8_APP.h	0	0	•
✓	DEMO9S08QG8_APP.c	5296	14	•
	demo9S08QG8.h	0	0	•
	Startup Code	132	6	•
	Prm	0	0	•
	Libs	12K	2K	•
	MC9S08QG8.h	0	0	•
	MC9S08QG8.C	0	94	•
	37 files	17K	2K	

Figure 5 Code warrior Project View and Debug button

Click the right pointing green arrow in the debugger menu at the top to start your program running. (The figure indicates communications selection which should not be necessary.) You should see the LEDs blink on and off together. When you press the each of the two pushbuttons, the LEDs will blink faster because the number being counted down will be smaller. If you press both buttons, the number will be so small you can’t see the blinking, though it would be detectable with an oscilloscope.

You can experiment with the debugger. If you stop the application from running, you can execute just one instruction at a time, and watch the variables change, and even the LEDs being turned on and off in very slow motion, one click at a time. You can also set a "breakpoint" in your code so that the program will stop there if it is running and gets to the breakpoint.

7.5 Write a "C" program to do the same thing:

The steps to do the application this way are similar, but we use a "higher level language", C, to write the instructions. This is actually easier, but we won't know first hand how the program works in terms of instructions, though we can actually see that using the debugger.

The process is similar to creating a new project earlier, but this time specify "C" language instead of assembly. There are a couple of extra steps for the wizard after "Processor Expert." Leave the default of ANSI C, Small, and None for C/C++ options, and say "No" to PC-lint.

The program will now be in "C" instead of assembly, and you will start with a file "main.c" to modify instead of "main.asm." Open main.c. In "main" we won't want to use interrupyts, so delete the #include <hidef.h>, the EnableInterrupts; statement, and the RESET_WATCHDOG(); statement.

Add code before the "loop forever" (for(;;){) for initialization:

```
unsigned int i;
  PTADD=0x00;  //All pins of port A to be inputs
  PTAPE=0x1f;  //Enable pull-ups on Port A
  PTBDD=0xc0;  //Bottom 6 pins input; top 2 pins outputs (LEDs)
  PTBPE=0x3f;  //Enable pull-ups on input pins
```

And then inside the main loop:

```
     PTBD=PTBD&0b00111111; //turn top 2 bits of port B to 0 (LEDs ON)
     i=PTAD&0b00001100;    //load i with PTAD bits 2,3 (0,4,8,or 12)
     i=i<<11;              //shift i left 11 (multiply by 2148)
     for(;i!=0;i--){}      //wait for i to count down
     PTBD=PTBD|0b11000000; //turn top 2 bits of port B to 1 (LEDs OFF)
     i=PTAD&0b00001100;    // load i from PTAD bits 2,3
     i=i<<11;              //multiply i by 2^11s
     for(;i!=0;i--){}      //wait for i to count down
```

When you are done, the file should appear as shown in figure 6 below. Note that this works pretty much the same way the assembly program did, but it is easier to express things in a more mathematical form.

Compile (make) and debug the same way you did for assembly language. When you enter the debugger, you can simply make the program run (by using the green "go" button. Or, you can execute one C statement at a time, or you can execute one assembly / machine language instructio9n at a time. Notice that the main debugger window shows the C code, and to its right a window shows the corresponding machine code. When you stop, the appropriate li8ne of code where you stop is highlighted. There can be several machimne instructions corresponding o each line of C code. They will not necessarily be easy to understand, either. Notice in particular the machine code that corresponds to turning the LEDs on and off. Notice how that the C code blinks the LEDs more slowly; it takes more instructions for each step in C than for assembly. On the other hand, it is much, much easier to create a program to do something in C than in assembly. If you really want to know exactly what you are doing, and control precisely how it is done, and want the utmost in speed, assembly might be worth considering. But for most practical applications people write in C or some other higher level language. C (and C++) are preferred languages for microcontrollers.

```
#include "derivative.h" /* include peripheral declarations */
void main(void) {
  unsigned int i;
  PTADD=0x00;  //All pins of port A to be inputs
  PTAPE=0x1f;  //Enable pull-ups on Port A
  PTBDD=0xc0;  //Bottom 6 pins input; top 2 pins outputs (LEDs)
  PTBPE=0x3f;  //Enable pull-ups on input pins
// Main program loop
  for(;;) {
     PTBD=PTBD&0b00111111; //turn top 2 bits of port B to 0 (LEDs ON)
     i=PTAD&0b00001100;    //load i with PTAD bits 2,3 (0,4,8,or 12)
     i=i<<11;              //shift i left 11 (multiply by 2148)
     for(;i!=0;i--){}      //wait for i to count down
     PTBD=PTBD|0b11000000; //turn top 2 bits of port B to 1 (LEDs OFF)
     i=PTAD&0b00001100;    // load i from PTAD bits 2,3
     i=i<<11;              //multiply i by 2^11s
     for(;i!=0;i--){}      //wait for i to count down
  } /* loop forever */
  /* please make sure that you never leave main */
}
```
Figure 6 C program to blink the LEDs.

7.5 Modify the "C" program to blink the LEDs in an overlapped 90 degree pattern.

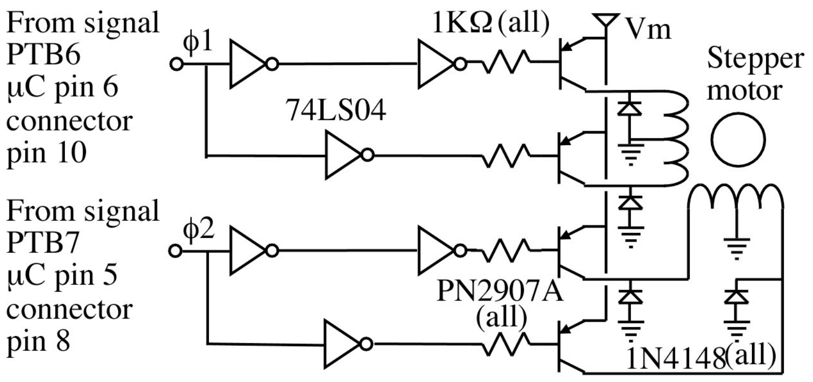
To drive a stepper motor we need square waveforms differing by 90 degrees. We can do this by going to four steps (LED's: 11, 01, 00, 10) instead of two (LED's: 11, 00). Only the code in the main loop needs to change:

```
     PTBD=PTBD&0x3f; //turn top 2 bits of port B to 0 (LEDs ON)
     i=PTAD&0b00001100;    //load i with PTAD bits 2,3 (0,4,8,or 12)
     i=i<<10;              //shift i left 11 (multiply by 1024s)
     for(;i>0;i--){}      //wait for i to count down
     PTBD=PTBD|0x40; //turn bit 6 of port B off (LED 1 OFF)
     i=PTAD&0b00001100;    //load i with PTAD bits 2,3 (0,4,8,or 12)
     i=i<<10;            //shift i left 11 (multiply by 1024)
     for(;i>0;i--){}      //wait for i to count down
     PTBD=PTBD|0xc0; //turn top 2 bits of port B to 1 (LEDs OFF)
     i=PTAD&0b00001100;    //load i with PTAD bits 2,3 (0,4,8,or 12)
     i=i<<10;              //shift i left 10 (multiply by 1024)
     for(;i>0;i--){}      //wait for i to count down
     PTBD=PTBD&0xbf; //turn bit 6 of port B to 0 (LED 1 ON)
     i=PTAD&0b00001100;    //load i with PTAD bits 2,3 (0,4,8,or 12)
     i=i<<10;              //shift i left 10 (multiply by 1024)
     for(;i>0;i--){}      //wait for i to count down
```

When you connect the two LED signals to an external circuit to drive a stepper motor, you should see the motor rotate at a speed that depends on the pushbutton settings.

7.6 Add a stepper motor

The circuit shown in Figure 7 can be constructed on a solderless breadboard to demonstrate how the signals to the LEDs can be used to drive a stepper motor. Three wires, Ground, PTB6, and PTB7 need to be run from the 20 pin connector on the DEMO board to the solderless breadboard. You need to provide 5 Volts to power the 74LS04, and that same 5 Volt Supply should also be used for the motor power supply Vm. (Note that we are using a "high end driver" so that the stepper motor power connection can be to ground. The four diodes protect the transistors from damage due to the inductive effect of the stepper motor windings.



From signal PTB6 µC pin 6 connector pin 10
ϕ1
1KΩ (all)
Vm
Stepper motor
74LS04
From signal PTB7 µC pin 5 connector pin 8
ϕ2
PN2907A (all)
1N4148 (all)

Figure 7 Stepper Motor Circuit

Warning: If you use a motor supply above 5 Volts, with this circuit, you can damage the 74LS04 inverter and possibly the microcontroller as well. For higher voltages, you need to use the 74LS06. Since that device is "open-collector," you will need to use pull-ups for the first inverter for each phase. You should also use a resistor (about 1K) between the base and emitter of each transistor to avoid leakage currents when the transistor is supposed to be off.

By pushing pushbuttons SW1 and SW2, the speed of the stepper motor can be changed. If both buttons are pushed, the frequency is so large that the stepper cannot function; it just vibrates and does not rotate. (The signals can still be observed on the oscilloscope.) You could modify the program, if desired, to provide for rotation in either directions (by reversing the phase relationships) or provide for more variation in speed. Some of these things will be done in a later lab exercise.

7.7 Report:

There is no report for this lab, but you do need to demonstrate success.