EGR247: Programming for Embedded Systems Initial Lab Demo

A lot of what was done in the first introductory session was similar to the Mechatronics (EGR222) Lab 7 exercise which also introduces students to the use of Code Warrior and the HCS08CG8 microcontroller. (This document was originally written for the QG8, and Code Warrior 6.x. Some adaptation may be needed for the SH8 and CW10. The main hardware difference is that the SH8 has an extra 4 pins for the C port, but there are some differences in some of the interface modules as well. One difference is the method used to "feed the dog" or, alternatively, shoot the dog. There are clocking differences too.)

Here are a few things that we took note of along the way:

1. "Include files" – in a c file (name.c "source" file containing your code), include statements in effect cause the referenced file to be treated as if it is actually in your file at the same place. As "main.c" is when it is created, there are two includes. One is to <hidef.h>. The symbols "<" and ">" indicate a file that is a provided file that generally corresponds to a library or some other vendor supplied functionality. The file is not physically located in your project folders. The best way to find the file if you want to look at it is right clicking on it and selecting "find and open file". (On my machine, the path is C:\Program Files\Freescale\Codewarrior for Microcontrollers V6.1\lib\HC08c\include\hidef.h.) Other commonly included files are <math.h> to give acess to a lot of useful math functions, <stdio.h> which lets you do I/O using ascii strings, and <string.h> for string manipulation utilities. We won't use most of these because their functions take up too much memory. We are programming a small machine.

The second kind of include file is referenced using quotes - "derivative.h" This means that the file to be included is a "local" one that either you create or Codewarrior creates for you when you set up your project. You will be creating "header files", those with the ".h" suffix, to include in your projects in the future. You will use quotes, and the file you create will be in your project. For derivative.h, the project path is (on my machine at home): "C:\Documents and Settings\Administrator\Desktop\EE147\EE147init\Sources\derivative.h." Notice the folder "EE147init" is the project folder that I will use to keep all the files specific to this project. Your project file may well be elsewhere.

If you open "derivative.h" you find the statement " #include <MC9S08QG8.h>." This file defines specific features for the particular microcontroller you are using. It's so important that the Codewarrior project setup includes it in the project even though it is not actually located in the project folders. (It's buried inside the CoreWarrior folders.) Notice the path to a given file is visible at the top of the window in which it is displayed within Codewarrior.

2. A second thing to notice right now is that the "Project view" of your project files at the left of the Code Warrior window is NOT the same as the organization of these files within the file system of the operating system. You can move files around in the project view and it has ne effect on where they actually are. There really is a "Sources" folder within your project folder, and your source files ".c" are there. But there is no separate

folder for your "include" files. They are right in the "Sources folder" along with the .c files. (I think CW10 does this a bit differently.)

It is possible to add source and include files to your project that are not even in your project folder. For example, you might want to use "terminal.c" and "terminal.h" but not copy the files into your project, but simply refer to them somewhere else. That means if you make a change to these files, you make a change to them for all the projects which include them. This can be tricky and dangerous. Unless you specifically want to do that, and live with the hazards and risks, I suggest that you copy all source and include files into your project and then add them, rather than refer to a copy elsewhere. Warning: a good many provided example code samples refer to files outside the example project folder. Watch out!

You add new files to your project by "new text file" on the file menu, save it in your sources folder, then add it to the project under the "Project" menu. You can then move it around in the project display to put it under "Sources" or "Includes" as appropriate. To add an existing file, put a copy in "Sources" then add it using "Add files" under the Project menu. You can also get rid of files by using "Remove" on the edit menu while the file is selected in the project view.

3. C functions allow you to pass variables in and they return one value. There is always a "main" function. It's where execution of your "C" code begins. The "main" function is in the file "main.c". That file could (and will later) include other functions as well as main(). [We will refer to functions using "name( )". What's actually within the parenthesese is the calling arguments, but when we refer to the function itself in discussion those are not detailed usually.]

The "main" function has the form:
void main(void) { ..... }.

The first "void" indicates that, unlike most cases, the main function does not return anything. In Unix systems, it does; it returns a 1 or a 0 to indicate to the operating system that the program (application) completed successfully or unsuccessfully. But we don't have an operating system. In fact, we don't want "main" to ever return. We want it to run forever. Not what you want on a PC.

The second "void" within the parentheses indicates that the "main ( )" function takes no arguments. That is, no information is passed to it when it is called, or invoked. Again, that's different from Unix. A Unix executable called from a terminal window may have several "arguments" (or "calling parameters") passed to it. These are passed to the program as a series of ascii strings, each corresponding to one of the parameters. So the "main" under Unix would look like:
int main(int argc, char *argv[]){ ..... }.

Here argc is how many parameters, and argv is a list of strings, each passed from the operating system. But we don't have an operating system. Yeah, things are different inside a microcontroller. Quite different.

4. So, inside "main ( )" you find two primary parts. The initial part is to do stuff right when you start up. Initially, the only thing "main ()" does is "EnableInterrupts." If you want to know how that is done, look in hidef.h. In fact, that's why this file "main.c" needs to include hidef.h. It's where the compiler finds out what to do with this statement.

We don't need (or want) to enable interrupts. But we don't want to forget about them. So we will "comment out" this statement.

Put a "/*" before and "*/" after the "EnableInterrupts;" statement. These delimiters are how C shows that something is a comment, and to be ignored by the compiler. (Because this compiler also does C++, it also respects the C++ convention of putting "//" at the beginning of a comment. If you do that, the rest of the line is comment. Since this is strictly C we won't usually want to do that.

If you wanted to, you could now also comment out the hidef.h include.

5. Now we will put in some code that will make Port B pin 6 (which is connected to LED #1) an output and turn it on. Right at the beginning of the program. We do that with a couple of C statements:

```
PTBDD_PTBDD6=1;
PTBD_PTBD6=0;
```

The reason we can use "PTBDD_PTBDD6" as a symbol is because it is defined in the file MC9S08QG8.h, which is included by derivative.h, which is included at the top of main.c. It's defined on line 284 of that file. It's a big file. The Port B data direction register is at address 0003 (hexadecimal), and we want to make the 6th bit a "1" in order to make that pin, named "PTB6", an output. This file is the key to finding the specific hardware resources of our microcontroller. On other microcontrollers, Port A and Port B may be elsewhere. Port B itself is at address 0002. So these C statements directly manipulate a couple of bits in memory that are actually directly attached to this one device pin that we can observe using the LED that is attached to it (assuming we have not removed the "strap". See the tables in the little booklet that came with youyr microcontroller kit to see where the other devices ate connected to your microcontroller. There are two user pushbuttons, two LEDs, a potentiometer, and a serial port. You can also get to any of the pins using the connector.

6. You can now "make" the program (compile and "link" it) and then "debug" it. The debug button downloads the executable code into the microcontroller and then starts it up. Notice that when you successfully "make" the project, the little red check marks in the project view go away. They indicate files that have been modified since the previous "make." Once you "make" the project, you will see that symbols defined for the microcontroller (and others defined earlier in the file) appear in a light blue color. This is a good way to tell that they are spelled correctly. (Notice also that "key words" in C are shown in a dark blue color, such as "#include" and "for".)

When you finish getting the program loaded in the debugger, it stops just as "main" is beginning to execute. It is stopped on the " PTBDD_PTBDD6=1;" statement. You can choose to make the program "Go" (start executing) by clicking the green arrow. But often it is interesting to watch it execute step by step, with the "single step" button. Before doing that, notice that you can see the corresponding assembly language instruction (in the window to the top right on the debugger): E092 BSET 6,0x03. This instruction is at address E092 (hexadecimal). It is the "Bit Set" (BSET) instruction, and it is to set bit 6 at address 3 (the place in memory where the data direction register for Port

Bis). So, here's a case where one C statement directly corresponds to one assembly level instruction. You can also look down on the left and find a window where you can observe the data memory of the microcontroller. You can open up the "PTBDD" item and observe wither the value of the byte stored there (in hex) or the individual bits. Right now the byte has the value 0.

So, if you click on the "single step" button, the statement is executed and that bit of the data direction register is changed to "1". That changes the value of the byte to 64 (decimal) or 40 (hex). Notice the red color indicating the changed data. The machine has now executed that instruction, and now has the program counter (PC) at E094, where the next instruction is to be found, and at the next C statement "PTBD_PTBD6=0. If you look at your physical Demo board, the LED comes on because now this pin is an output. It is a "0" (low) so the pin sinks current that is drawn through the LED.

The next instruction actually doesn't change anything, because the value of Pinb 6 is already 0; it is set to 0 at reset.

If you continue executing, you get to the "for" loop. A more typical for loop does something a give number of times. For example:

for(i=0; i<4; i++){ .....stuff..... }

The first part inside the parentheses is executed once at first when the statement is encountered. here it sets the variable "i" top zero. The second part makes a logical check at the beginning of the loop. If i is less than 4, the stuff that follows is executed. It will be the first time, because i starts off as zero. In this case, "stuff" happens 4 times. The last part is what happens after the stuff is executed. "++" means increment by 1. So, 1 is added to the variable i before it is checked for the next iteration.

In this case, we see in our code " for(;;){ ..... }. That means do nothing on entry, make no check to see if you should exit, and don't do anything at the end of the loop. In other words, "do forever." That's exactly what we want. What is inside the loop is what we will execute over and over again until the microcontroller gets a "reset" or it loses power. In our simple initial program, what it does is "__RESET_WATCHDOG().

The "RESET_WATCHDOG( )" function is in all caps indicating it's not really exactly a function in the usual sense. We won't go into this now, but if gets translated into the assembly / machine code "STA 0x1800". Address 1800 hexadecimal is a special register that, when you write to it, it "resets" the "watchdog timer." The watchdog is a timer that, if it doesn't get reset, will reset your machine the same way that hitting the reset button will. (In some microcontrollers, including the HCS08JM series, the watchdog is not enabled unless you enable interrupts. On this one, it is automatically enabled. So, what you are doing is called "feeding the watchgdog" or just "feeding the dog" so it doesn't bite you. The thing that you can do instead is shoot the dog, that is, disable it. That can be done by a statement "SOPT1_COPE=0;" or SOPT=52 (?). (I need to check this.) (NOTE: The SH8 watchdog is different! Consult the SH8 manual – download.)

Notice that the instruction after the "STA" is given as "BRA *-3". This means branch to this addresss minus three. That takes the PC (the pointer to the next instruction) back to the STA. That's how the endless "for" loop is implemented. (The actual BRA instruction codes the "-5" in 2's complement hexadecimal, since the displacement is from where the PC woiuld be if it hadn't branched. So it is really the bit pattern "FB" or "11111011".

6. You can create a loop where the LED will blink on and off (too rapidly to see) by leaving the data direction statement where it is, but moving the Port B data statement (and adding another one) into the "main loop." After the "for" you then have:

```
PTBD_PTBD6=0;
PTBD_PTBD6=1;
```

After redoing your make, debugging and replacing the earlier program, when you "run" the LED is a little dim. When you stop the debugger, the LED may be on or it may be off. Rather random. (It actually depends on your reflexes. See if you can make it flip just once. Ha!) You can do "single step" and see the program go one step at a time, and see the LED go on and off.

7. Now, to make things more interesting, we add a "variable" to count each time we go through the loop. Add a new statement "short i=0;" right at the beginning of main( ). It must come before the first executable statement that sets Port B data direction. This statement declares, "Let there be a short (16 bit) integer variable known as 'i' and let that variable initially have the value 0."

Then, inside the loop, add a statement between the two bit setting statements for Port B to increment i. Here's how main.c looks after doing so:

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

void main(void) {
  short i=0;
  /*EnableInterrupts;*/ /* enable interrupts */
  /* include your code here */
  PTBDD_PTBDD6=1;
  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
      PTBD_PTBD6=0;
      i++;
      PTBD_PTBD6=1;
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Now, when debugging, it is interesting to see that in order to increment the variable i (which you can see changing in the lower data window), it takes 3 machine language instructions. Also, i is not initially actually 0; it takes a few instructions to create "i" on the stack and set the X register to point to it. After doing that, i is set to zero by "CLR" (clear instructions) for each of the two bytes, and incremented (later) with the sequence:

```
INC 1,X  (increments the least significant byte)
BNE *+3  (skip the next 3 byte instruction if that increment didn't get a zero)
INC ,X   (increments the most significant byte)
```

Notice that the BRA at the end of the "for loop" now has to go back -12 instead of just -3. That's because there is more stuff inside the loop.

Because of the time needed to increment i, the on and off cycles are more nearly balanced.