EE 342 Microcomputer Based System Design
Fall, 2018

Potentially Useful Texts (not required):

Fabio Pereira, *HCS08 Unleashed, Designer's Guide to the HCS08 Microcontrollers*, 2nd ed., 2009, ISBN 1-4196-8592-9 (From EE247)

Ivor Horton, *Beginning Visual C++ 2008*, Wrox / Wiley Publishing, 2008, ISBN 978-0-470-22590-5 or Horton, *Beginning Visual C++ 201*2, Wiley, 2012, ISBN 978-1-118-36808-4 or 978-1-118-43941-4, 978-1-118-41703-4, 978-1-118-43431-4. (I don't know why the book gives 4 different ISBN's.)

Various NXP/Freescale HCS08SH series microcontroller documentation, available from www.nxp.com, (formerly Freescale).

Jan Axelson, *USB Complete*, 4th ed., Lakeview Research, 2009, ISBN13 978-1-931448-08-6.

Scheduled class times: TR 9:30-11:20 SLC238
Instructor: John B. Gilmer Jr. Office hours: TBD Office:SLC220 Phone: x4885
Pre/co requisites: EE241 Digital Design, EE247 or CS126 or equivalent (see instructor)

Background:

Computers are everywhere. Small, inexpensive computers have found their way into almost every electrical product, and add valuable functionality at relatively small cost. Many do so without our being aware of them. Others are so ubiquitous that we don't even notice. For example, the control of a microwave oven involves pushing buttons and selecting time and power levels, something that could be done with a mechanical switch and timer. And was, in less expensive models, until recently. But now, the computer (costing less than $1) that does this control allows doing it cheaper than with the mechanical timer and switches, and allows inclusion of a clock as well. In the future, higher end microwave ovens may tie into your computer or your house's control system by wireless to add a startup when your alarm goes off. Adding an embedded computer is part of the development of many products, and this trend will only accelerate, with the "Internet of Things" and other recent developments.

So, this course is focused on preparing the student to participate in the opportunities that knowledge of this technology will open up. The components include the hardware of the relatively simple computers that are used as microcontrollers. But, these seemingly simple machines are actually quite sophisticated. We will only be able to study the core parts and a sampling of the interfaces and techniques available.

We will also be studying and developing software. Or, when it is embedded in a product, "firmware." This will require a good bit of understanding of software fundamentals. We will be using machine language (to a limited extent) and the "C" and "C++" programming languages. This is not a "programming" course, and we will not delve into the more complex aspects of programming. Rather, we will use "just enough" programming to meet our needs. That's why EE247 or CS126 is a prerequisite. But, that "just enough to meet our needs" is still quite a bit.

Most important, we will be focused on the relationship between the software and the hardware on which it runs, and with which it interfaces to the outside world. For, embedded computers do not just run games that communicate with keyboards and monitors. They sense things in the real world, and they cause things to happen. This means using digital and analog interfaces to electronics and physical devices like photodetectors, voltage or current sensors, motors, and LED's.

Also important is that microcontrollers often need to communicate with other computers. We will be studying and using the ubiquitous "USB" port for this. Since we are communicating to another computer (in our case a Windows PC) we will also need to look at programming at the PC "host" end of the communications.

The microcontrollers chosen for this course are the NXP (formerly Freescale). We used to use the HCS08JM60 and its more capable "Coldfire" series counterpart. But these products have been phased out, replaced by the "ARM" processors with similar function. So, what we will do is start with the same S08SH8 pmicrtocontroller and demo boards that were used in EE247, Programming for Embedded Systems, as we deal with fundamentals, then shift to a different microcontrtoller that has a USB port. (That shift used to be to the MCF51JM128, a "Coldfire" V1 that had the same pinout interface as the simpler S08JM60.) We will, at least initially, develop programs to run on the microcontroller using the "Codewarrior" development software. For Windows 7 or later, you'll need to download a more recent version 10.x. The microcontroller comes on a small PC board that includes a variety of interfaces and devices, including LED's, pushbuttons, a pot and optiocal sensor, and debugging support. (The software and documentation can be downloaded directly from the NXP web site.)

Ultimately, we will shift to a USB interface using a more capable processor. That should be chosen by the end of September. I'm in the process of getting some samples which I'll try out, and from them choose a product that will be both suitable and hopefully also minimize the complexities of the transition. Using USB is considerably more complicated than serial; ports, and we want to see that working. We will build toward a final project where each student's microcontroller will control part of an HO gauge train set, with the microcontrollers coordinating with each other, and with the students via their host PC's.

So, why USB? You don't find simple serial ports so much anymore; everything seems to have gone to USB (or other) interfaces such as Ethernet. The need to focus on this shifts the emphasis more in the direction of software. However, the critical issue remains, that the hardware and software must work together to get the job done. You need to develop a fundamental appreciation of both, and how they work together. That is our emphasis.

(Note: Normally this course is offered in the fall of even numbered years.)

Schedule: (Some adjustments are likely)

Week of:	Topics covered	Reading, Tests
1 Aug 28	Overview and Introduction	Terminal, SH8 man.
2 Sep 4	Programming basics (continued) HCS08	Terminal SH8 man.
2 Sep 11	More programming, basic interrupts and I/O	SH8 man., examples
3 Sep 18	Timers, clocks, and Pulse Width Modulation	SH8 man. Report#1
4 Sep 25	Buffering, timing issues, priorities	
5 Oct 2	Sensing things, controlling things	test #1 (SH8 based)
6 Oct 9*	Power handling, static, noise, and Voltage hazards	handouts. Report #2
7 Oct 16	The 32 bit (ARM?) processor (specifics TBD)	
8 Oct 23	Communications basics, use USB as serial port (?)	Report #3
9 Oct 30	USB ports, Interrupts, timing with USB	
10 Nov 6	Visual Studio, Applications intro	Horton
11 Nov 13	Visual Studio, Applications (continued)	test #2 (electr, control)
12 Nov 20*	Host software	Horton
14 Nov 27	Host software, project demonstration (to be scheduled)	Horton
15 Dec 4*	Summary, misc. topics	-
16	Exam	(comprehensive)

* indicates a "short" week.

About the books: These are not normal “textbooks.” They are reference books, and will be used mostly in that manner, rather than as textbooks are normally. As such, the books mentioned (Pereira and Horton) should not be terribly expensive. But, they are not required. We will mostly be referring to technical data available online describing the microcontrollers we are using. The Fabio book includes some help for using Codewarrior (the development software suite), which is nice. There is nothing in the book that you cannot find in various documentation available from NXP and elsewhere, but it’s nicely collected and presented.

The Horton book is on the list for two reasons. First, it has some good introductory programming stuff for using C and C++. Second, it has help for using Visual Studio 2008 (or 2012), Microsoft’s software suite for programming the PC. The Visual Studio 2012 version is more recent, more expensive, and more useful if for some reason we wend up using Visual Studio 2012 which is what I’m expecting. Now Visual Studio 2015 (I think it is) is out as well. I believe both 2012 and 2015 are in the labs. The earlier version of Horton (for 2008) ought to be available cheaper. However, if you run stuff on your own PC, you ought to go with 2012 or later. You may be able to get by fine without this book, especially if you are comfortable with C++ and programming under Windows. We need C++ to code on the PC end a program (application) that will communicate with our microcontroller.

The USB book is included as a “useful reference” because we are going to dive in and try to understand at least the essentials of how USB works. My guess is that it’s not adding enough to be worth the purchase price unless you are really interested in the internals of USB, and the book doesn’t quite give enough to fully satisfy. But, it’s the

best I've found. I'll put my copy in the lab. In addition, you will want to download, and print selected sections from, certain reference materials from NXP. Many of these reference documents can also be found on the CDs that comes with the microcontroller DEMO boards (or are designated downloads). In particular, we will need that as a source of detailed information on both processors.

Each student will receive a "Demo" board for the S08SH8 microcontroller. Supplementary documentation is available at the NXP web site. But we are going to start with a simple "terminal" program (that is based on software supplied for the now-obsolete JM version of the S08 family). The terminal program allows the microcontroller to communicate back and forth with a terminal emupation program (e.g. PuTTY or Hyperterminal) on the PC. The terminal program executes user defined functions on the microcontroller to do things such as blink LED's, run motors, tell the time, or measure Voltages. That's a convenient starting point for considering microcontroller applications that need to communicate with the outside world.

Lab schedule: We will start with the terminal exercises, then do a series of projects that will support the final project, which will be the HO train exercise. For that one, we will have a party and invite friends and family, for "The Running of the Trains" at the end of the semester. It will be fun. We'll figure out a time when it will best suit guests (family and friends) you may want to invite.

Expected lab Exercises (the dates assigned are tentative and subject to change):
1. The terminal program initial exercises "getting started"
2. Text based interaction, using peripherals (A/D, PWM) and interrupts (serial port)
3. Interacting with the real world: sensors and actuation
4. Interacting with the user via USB and your PC application (on TBD 2nd uC)
5. Performance assessment (benchmarking) (If we can get to it)

The project:

The last big exercise is "The Running of the Trains." Each student will be responsible for a part of an HO train layout, typically two pieces of track and a switch, that the student's microcontroller will control. Our goal is automatic cooperative control, with each student's computer interacting with those adjacent to communicate about the state of the railroad. The goal is to keep the trains moving: Route arriving locomotives onto available tracks, and hold them up when the next section of track is blocked. (We won't try to run whole trains; just doing the locomotives will be challenge enough.)

We will do this in SLC238. The track is mounted on boards about 1ft x 6ft (or 8 ft) long which can be connected end to end. We will have to figure out how to configure the overall system when we know how we can set up our track and stations in a way that does not interfere too much with other things (if any) happening in the lab. With 9 students this offering, we will have to conficure the track in a "U" shape moving around the lab across the back. We will want to determine who has what before Lab 3, when we will start to control things on the railroad. The last few weeks of class will be dedicated to bringing this project to a state of completion.

Grading: Two tests, final exam, about 5 lab reports. Reports will be of limited formality – abstract, documented code, schematics if appropriate, maybe some tables or screen shots, and conclusions. The Final Project will have a formal report. No graded homeworks. Maybe one or two pop quizzes.

Two tests at 10%: 20%
Five lab exercises at 5% each: 25% (or 4 at 6%)
Final exam: 25%
Final Project and Report: 20% (or 21%)
Pop quiz(zes): 2%
Class participation: 3%

Windows 7:
The lab computers currently run Windows 7. To be compatible with Windows 7 (at least in 64 bit mode), you need to find and download Codewarrior 10.x from the NXP web site. Even then, there may be some issues. I will have to check these out and will document what is needed when we select our USB microcontroller.

Web Page:
I will be posting materials to support this course at < http://www.jbgilmer.com/EE342/EE342.htm>. The stuff there as of this writing is from 2 years ago; I'll be replacing it and adding additional documents as we get to things.

Conclusion:
I believe this will be a very worthwhile course. You will come out of it with some skill in programming with C and C++; you will have done programming under Windows with Microsoft Visual Studio and C++; you will know how to use and program a microcontroller, and you will be able to say you have programmed a USB interface. You won't know everything you'd like to know, but you'll know how to get started on microcontroller projects you may meet in the real world.

EE342 Microcomputer Operation and Design
Laboratory Exercise #1

Demo Sept 13; Report due Sept 20

Objective:

This lab exercise is intended to help get familiar with the tools, reference documents, and some of the most basic features of the microcontroller and the terminal program.

Preliminaries:

Read up on the HCS08SH8 a bit, and the demo board. In particular, figure out what ports and bits of those ports go to the various LED's, pushbuttons, sensors, and the serial port. (The schematic is a big help for that.) Look through the various source and header files of the terminal project in CodeWarrior, and the supplied documentation, and get used to how things are organized and accessed.

Try the terminal program on your own (see reference) and see if you can get it working. We will make sure everybody gets to this point in lab, probably the first week. See the appendix "Initial Lab Exercise: Terminal Program" at the end of the syllabus.

Procedure:

1. Get your files set up and ready

Create a new project for the terminal program, and copy in the needed files. (See the terminal documentation I supplied.) Adding files to a project can be tricky. You want to have the file in the sources or headers folder (as appropriate) and you also want it to be properly accessed in the "project view" of Code Warrior. If you want to add additional files, I suggest:

For each file to go into the project that is NOT in the source folder already:

1) Open the file by double clicking it in the project view window.
2) "Save as" (file menu) to your project source folder.
3) Close the original file. (but it's still selected in the project view)
4) Delete the file from the project (on the Edit menu). It will ask if you really want to do that; you do.
5) Add a file to the project (Projects menu) and point to the new copy in the source folder.

Now, this isn't quite enough. Look at each source file, and see if there is a "# include" statement that refers to a file that is not local. For example, if you see anything other than "#include "<file_name>" ," that is, paths to other folders, you wan to track down the corresponding "include" (.h) file and also put a copy in your "sources" folder. I like to formally add it to the project, too, so that it's on the list and therefore easy to open and examine and modify.

You also need to modify the code to change the "#include" in all of your code so that it refers to your local copy.

At the end of all this, you should be able to "make" an executable and then "debug" and run it by clicking the green debug arrow and doing the same things with the terminal emulator (PuTTY or hyperterminal) as in the introductory lab.

2. Modify the code to control both of the LED's, and to monitor both pushbuttons

This has several different parts to it, listed below:

1) Change the code in "target.h" so that you set up the ports for the other LED's and pushbuttons. Find the init_board function. Look at the code to set up the LED's. Make sure it works for all eight. (Be sure you understand what's going on here. Make reference to Chapter 6 of the HCS08JM60 manual.) Now look at the code right underneath for the pushbuttons. You want to be able to read all four. Modify the code accordingly. After doing all this, do a "make" to ensure you have not inserted compile errors.
2) Modify the existing cmd_led function in cdc_main.c to handle all of the LEDs. To do that, you will need to modify the command to see which LED the user wants to flip. So, when the user gives the command "Led 2" you flip the 2nd LED. That means reading the parameter. Add new local variables int "x" and "n" to the function. Then, you can use the standard I/O function: "n=sscanf(param,"%d",&x);" to read the value of x from the character string param. (Or, you could try x=param[0]-0x30.). If you use sscanf, you will need to put "#include <stdio.h>" in among the includes. Now it's just a matter of putting in code to flip the appropriate LED. (You could have 8 variables to save the states, or use bits in one variable, or even read the current state from the I/O register.) Try your code and see if it works. Make, debug, and run the terminal emulator to test it.
3) Add a new command to test the switches. Look at how the LED command works, and do likewise. You don't need the param. You will print a string that reports the state of the pushbuttons. There are several ways to do that. You can use "sprintf" to print a message to a string, then pass the string to the print(string) function. Inside your command, you will need to test each pushbutton as you put your message together. (Here's a potential problem: If you were doing this with the SH8, there's not much memory, and using these sscanf and sprintf uses up a lot of it. You may run out at some point. That won't be a problem with larger uC's later.) Look at the project.map file to see where in memory everything is, and how much space each function uses.)
4) Add code in main to monitor the switches, and initiate a message out that reports whenever there is a change. This is code you would put inside the main loop after the call to terminal_process();. Run it and see it work!

4. Report your results.

This is an informal report. Include your modified code, a copy of main.c, and copies of anything else you modified. Also, include a transcript of a session in the terminal emulator showing a "help" command, then several commands to flip LED's and sample pushbuttons. Finally, write a "conclusions" paragraph as necessary to say that it worked (or didn't) and remark on anything of interests such as peculiarities, innovative things you did, problems encountered, or things that are still not doing what you want.

We will spend 2 lab periods on this stuff, and the report will be due at the following class.

EE342 Microcomputer Operation and Design
Laboratory Exercise #2

Demo by Oct 4; Report due Oct 9

Objective:

This lab exercise will extend the terminal based program of Lab 1 to utilize timing/clock and A/D resources, and to become familiar with basic interrupts.

Preliminaries:

Read up on the HCS08 interrupt. Clock/timer. And A/D material. Study the demo board schematic and find out what signals to monitor with the A/D converter. Choose a single bit to use for a PWM Voltage output. You might also want to go back and look at the digital speed control lab exercise from the Mechatronics class.

Create a new project for this lab. I suggest deriving it from the previous project.

Procedure:

Do each of the following steps, pausing to compile and make sure your project works after each:

1. Add commands to read the potentiometer and photosensor. Be able to read an external pot as well.

1) Figure out how to initialize the various signals and ports to allow reading the 3 signals of interest with the A/D converter.
2) Add commands to read the potentiomenters, and to read the photosensor. In response to the command, print the Voltage to the screen. Use 12 bits of resolution. (Wire the external pot to 3 pins on the demo board connector.)

2. Add a clock to your project

1) Create new files clock.h and clock.c for your clock code. Plan to use a routine "(void) clock_init(void)" to initialize the clock, and a clock routine "(void)clock_update(void)" that will actually add to the clock at each interval. You can put the clock commands in main() or in clock.c. The latter is probably preferred, but the former is easier. Create global variables for the clock data (milliseconds, seconds, hours).
2) Put code in "clock_init()" to initialize the real time clock. Since we will want to use the clock for timing the PWM waveform, we'd like high resolution. I suggest 10 mSec, at least for a start. You may try to use faster timing later.
3) Write code for the clock update. At first, make this clock_update() routine code that is called from within main() inside the loop. Call it after the other things in the main loop, if you see that the timer is indicating it's time for an update. (Making the update resets the timer.) Clock_update should maintain the time in milliseconds, seconds, and hours. (Don't bother with days!)
4) Add new commands to set the clock (to a specific time) and to read out the time.

3. Make the clock clock interrupt driven.

1) Modify your initialization to enable the clock (RTC) interrupt.
2) Modify clock_update to make it an interrupt service routine. To do so, start the function as below:

```
/*************************************************************
Interrupt Function name: rtc
Note: Interrupt service routine for RTC module.
*************************************************************/
interrupt 29 void rtc(void){
```

3) Delete the polled call from main();

4. Add a Voltage command controlled PWM output

1) Add a new command to set an output Voltage. (Enter with resolution of at least x.x Volts.)
2) Add a new global variable for the output Voltage.
3) Add a function called from the clock which varies a PWM waveform to some output pin so that it will have the average voltage given by the command. With a clock running as slow as 10mSec, if the waveform is 10Hz, the resolution will only be ½ Volt.
4) Try changing the clock to allow better resolution, and/or a faster frequency.
5) Observe your PWM waveform on an oscilloscope for a couple of different settings.
6) Optional: Drive a PM DC motor, or use the PWM module to do it.

Report your results:

This is an informal report. Include your modified code, copies of main.c, clock.h, clock.c, and copies of anything else you modified. Also include a transcript (or transcripts) of a session with the terminal emulator showing a "help" command, then several commands set and read the clock, read the pot at a couple of different values, and read the photosensor for a couple of different illumination levels. Also include in the terminal emulation session your command for an output voltage being given. Give a sketch of the PWM waveform for the couple of different settings tried. Finally, write a "conclusions" paragraph (or as necessary) to say that it worked (or didn't) and remark on anything of interests such as peculiarities, innovative things you did, problems encountered, or things that are still not doing what you want.

EE342 Microcomputer Operation and Design
Laboratory Exercise #3

Demo Oct 17; Report Oct 24

Objective:

This lab exercise will extend the terminal based program of lab 1 (or 2) to control an HO train track and monitor track conditions.

Preliminaries:

You might want to create yet a new project for this lab. I suggest using the previous project as a starting point.

Procedure:

Do each of the following steps, pausing to compile and make sure your project works after each:

1. Build, and test, track driver electronics

1) Build high and low end drivers for each rail able to handle about 2 Amperes. Together, these will control one rail, so you need two such pairs for each section of track. Include electronics that will prevent both high and low drivers from being on at the same time (for extra safety). The track driver pair should be capable of being put into "three state" (rails undriven). In the end, the section of track should support movement of a locomotive in either direction, or being floated. Test your track drivers unconnected to the microcontroller.
2) Build solenoid control electronics that will flip the switches on your pieces of track. It would be nice if there is some protection against the driver being on continuously. (There are several ways to do that. Consider using the "watchdog.") Test your solenoid control circuit.

2. Add sensors for detecting track Voltages and objects

1) Design, build, and test an electronic circuit to indicate that a high Voltage is on a floated section (rail) of track. This would be indicative of a locomotive arriving from an adjacent track section. You would like a TTL level logic 1 or 0 indicating this is the case or not. Similarly, design and build a circuit that will detect a low Voltage. (Your "floating" track section should have some large resistors that will keep its Voltage somewhere around 1/2 of the motor Voltage when undriven, so that you detect neither a high nor a low Voltage.)
2) Design and build an optical detector which will detect the presence of a locomotive somewhere on or approaching your sections of track.
3) Desirable but not required: Design a circuit that will detect an "overcurrent" condition (for example, a short circuit) on a section of track. This should be a small resistance with an amplifier to detect when the Voltage drop exceeds some threshold value, perhaps 3 or more Amperes.

3. Add commands to your terminal program to run the track.

0) Figure out what pins of your microcontroller to use for inputs from sensors and outputs to the track drivers. Suggestion: you might want to use PWM

capable pins for some track driver signals to allow speed control later. But, speed and direction could be separate signals for each track rail pair.

1) Add commands for each piece of track to make a locomotive on that track go forward, reverse, or stop (three state).
2) Add commands to flip each switch in one way or the other (but not leave power on!).
3) Add command(s) to read the state of the sensors.

4. Connect together and test it.

1) Figure out how you are going to arrange for power needed for your electronics. Be very careful about power you get from the microcontroller board; you don't want to blow that away. It should be practically impossible to connect that to 15 Volts accidentally.
2) Connect your track drivers to your microcontroller and test, demonstrate them.
3) Connect the solenoid drivers and test / demonstrate them.
4) Connect and demonstrate your sensors.

Report your results:

This is an informal report. Include your modified code, copies of main.c, and copies of anything else you modified. You should document your code well. Also include a transcript (or transcripts) of a session in the terminal emulator showing a "help" command, then several commands to manipulate the track and solenoids, sense things, etc. Annotate the transcript to say what was observed when the command was issued. Finally, write a "conclusions" paragraph (or as necessary) to say that it worked (or didn't) and remark on anything of interests such as peculiarities, innovative things you did, problems encountered, or things that are still not doing what you want.

EE342 Microcomputer Operation and Design
Laboratory Exercise #4

OLD VERSION! TBD – This will change. The Windows side will likely stay the same.

Objective:

Introduction to programming under Windows and communications via USB to the microcontroller.

Preliminaries:

Copy the "PC side" Visual Studio project for the hid demo to the Visual Studio 2008 (or 2012 or 2015) projects folder (inside "documents" folder). Copy the various files from the Windows Driver Development Kit into the hid libs folder inside the project folder. Open the project, and let it convert it to Visual Studio 2008 or 2012/15. Then, run the project (using "debug" in Visual Studio). You first need to reload the "hid demo" into the microcontroller. That is the program that came installed. To start the "generic hid" version, hold down switch 2 as you release "reset". With that running, "debug" the PC end Visual Studio code and you should be able to turn the 4 low order LED's on and off, and see the status of the two low order pushbuttons.

Once this is working, make a backup copy of the project (at each end) so that you can make modifications and yet be able to go back to the old version if need be. For the microcontroller, you should also copy needed files into your project folders, rather than modify the ones that are "common" to other projects, just as when we got started with the terminal program.

Project Goal: Modify the generic hid demo to:

1. Control all of the LED's and report all of the switches
2. Report back to the screen the potentiometer setting

This is a big project because it involves diving into the PC end of things. Step 1 is not all that hard because you are just doing more of the stuff that's already there. You can use exactly the same USB stuff. But for step 2, we need to be able to use an Edit field in Windows, and we need to modify the USB report coming back from the uC to the PC to be able to pass more than just a byte. So, in this one we are exploring a lot of new things.

Procedure:

1. Add the rest of the LEDs and switches.

a. This means first going into the microcontroller code and providing similar code for the rest of the switches and LED's, which is pretty straightforward. You will (probably) just modify hid_generic.c

b. Next, modify the PC end. First, open up the dialog resource and copy LED buttons and switch buttons. You might want to make the dialog box bigger, too. Note the names "IDC_CHECKn" for each new button. Change captions appropriately.

c. Then, go into the Visual C++ dialog header file and add new objects to the "dialog" in the dialog header file) "<project name>Dlg.h" for the needed cButtons. You also need member functions for the LED buttons that the user will push, that are called by Windows when that happens.

d. Modify the code in the dialog code file, "<project name>Dlg.h". You need to add to the Dialog's DoDataExchange() function and MESSAGE_MAP to associate the resource with the corresponding objects and functions in your dialog.
e. Modify update_leds() to accommodate the four new LEDs.
f. Add the new functions that call update_leds().
g. Provide for updating the switch indicators in a manner similar to what is done for the other two switches.
h. In your `OnInitDialog()` function you might want to open an output debug file into which you can write stuff for debugging help. Add writes here and there to write into the file when things like LED updates, switch pushes, and such happen.
i. Run it and debug to get all of the switches and LED's working.

2. Add an edit box to report the potentiometer value.

a. You need to add code to the generic_hid to sample the A/D converter to get the potentiometer value. Do this whenever a switch is pushed or unpushed. You will send the 12 bit value back to the PC.
b. In order to include the pot value, you need to make the message back to the PC 3 bytes: 1 for the switches as at present, and 2 more for the pot value. Change the message size to 3 bytes, and now you need to make the message not just an hcc_u8, but an array of three hcc_u8's.
c. You also need to go in and change the report description and size in the hid_usb_config.c file. Find the array that defines the message going from the microcontroller to the PC for the generic interface, and add two bytes (or more?) of report to the description. (I did the full 8 bytes allowed, since I was also sending additional data. Here's a copy.)

```
/* Modified for 8 byte message */
const hcc_u8 geh_report_descriptor[60] = {
    0x06, 0x00, 0xff,              // USAGE_PAGE (Vendor Defined Page 1)
    0x09, 0x01,                    // USAGE (Vendor Usage 1)
    0xa1, 0x01,                    // COLLECTION (Application)
    0x05, 0x08,                    //   USAGE_PAGE (LEDs)
    0x09, 0x4b,                    //   USAGE (Generic Indicator)
    0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
    0x25, 0x01,                    //   LOGICAL_MAXIMUM (1)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x07,                    //   REPORT_COUNT (7)
    0x91, 0x02,                    //   OUTPUT (Data,Var,Abs)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x01,                    //   REPORT_COUNT (1)
    0x91, 0x03,                    //   OUTPUT (Cnst,Var,Abs)
    0x05, 0x09,                    //   USAGE_PAGE (Button)
    0x19, 0x01,                    //   USAGE_MINIMUM (Button 1)
    0x29, 0x04,                    //   USAGE_MAXIMUM (Button 4)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x04,                    //   REPORT_COUNT (4)
    0x81, 0x02,                    //   INPUT (Data,Var,Abs)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x04,                    //   REPORT_COUNT (4)
    0x81, 0x03,                    //   INPUT (Cnst,Var,Abs)
    0x05, 0x01,                    //   USAGE_PAGE (Generic Desktop) /jbg/
```

```
    0x09, 0x36,                    //     USAGE (slider)         /jbg/
    0x15, 0x80,                    //     LOGICAL_MINIMUM (-128) /jbg/
    0x25, 0x7f,                    //     LOGICAL_MAXIMUM (127) /jbg/
    0x75, 0x08,                    //     REPORT_SIZE (8)   /jbg/
    0x95, 0x07,                    //     REPORT_COUNT (7)  /jbg/ maximum
    0x81, 0x02,                    //     INPUT (Data,Var,Abs)   /jbg/
    0xc0                           // END_COLLECTION
};
```

d. Now when you open up the usb port at the PC end, you should see that there are two more bytes of data. (Not a bad idea to check that with the debug output file.) Change the incoming report to make sure it's 3 (or more) bytes now.
e. Add an "Edit box" to your dialog using the resource editor. Note its identifier.
f. Add the cEdit object and needed functions to your dialog header file.
g. Add the needed entried to your data exchange and message map in the dialog cpp file.
h. Modify the code that updates the switch boxes when a message comes in to also get the 0-4095 integer from the message, convert it to floating point, and display it in the edit box you have added. Things to beware of: The HCS08 is big endian, the PC is Intel, hence little endian. The string displayed in the CEdit is Unicode, not ascii; you need to convert by padding a 0 in the top byte of each character. Be sure to select the previously displayed text before you replace it, otherwise you will just keep adding text. (Not a bad idea, though, if the idea is to display a record of changes!) (Later: we want to use Coldfire.)
i. Debug and demonstrate it.

Extra: Report not just the Pot, but also the X,Y,Z of the accelerometer.

Report your results.

This is an informal report. Include your modified code, showing copies (or well identified excerpts) of anything else you modified. If you can, take a screen shot of the running program on the PC. (I have not figured out how to do that on a Mac running Windows; it doesn't seem to have the needed keyboard key.) Finally, write a "conclusions" paragraph to say that it worked (or didn't) and remark on anything of interests such as peculiarities, innovative things you did, problems encountered, or things that are still not doing what you want.

Reference:

See "Guided tour of a generic HID USB Windows Application," J.B.Gilmer, Wilkes University Oct 30, 2010 (handout)

Get the "Windows driver development Kit" files needed (with modifications). These will probably be distributed by email.

EE342 Microcomputer Operation and Design
Laboratory Exercise #5

This will change, and we might not do it at all.

Objective:

Introduction to performance assessment.

Preliminaries:

Be sure that the Coldfire V1 support is installed in your computer. Do the series of exercises found in the "DEMOJM Lab Supplement" for the 32-bit Flexis JM128 if you have not done them already.

Project Goal: Compare the performance of the two processors:

The goal is to come up with a factor by which the JM60 and V1 Coldfire differ in speed. Each student will use a different algorithm, so that across the class we will have several points of comparison. The different exercises will then let us compute an "overall" ratio. We would also like to get a measure of performance in terms of "instructions per clock cycle".

Procedure:

Write a segment of C "benchmark" code to perform a simple function. The algorithms to be done by various members of the class are below. Pick something that nobody else is doing. Start with your old Lab #2 code and modify it.

a. Dot product, using iteration, with two long integer vectors of length three.
b. Cross product, for two long integer vectors of length three.
c. Product of two complex 32 bit fixed point (at radix 16) variables.
d. Floating point (32 bit) multiply and add.
e. Sort of long integers in a vector of length 4.
f. Perform a Fast Fourier Transform (FFT) of size 8.
g. Monte Carlo method for calculating the value of pi.
h. Numerical solution of the Schrodinger wave equations for Lithium (element 3).
Do each both simply and with "loop unrolling" or other optimized methods.

In all of these cases, write a function that performs the benchmark algorithm, with the arguments passed by reference. For example:

a. long dot(long *A, long *B, long n);

where A and B are previously defined as:

```
long A[3]={1,2,3};
long B[3]={4,5,6};
```

Note that since the algorithm works by iteration, the length of the vectors needs to be passed in. The complex numbers are actually vectors of length two. Come up with a reasonable assortment of values with which to computer the results.

Once the benchmark function has been written, write a terminal command that will exercise the benchmark. That means there needs to be a way to time how long the benchmark takes. Generally that is done by running the benchmark code repeatedly many times. One checks the time before, and one checks the time after, and the difference is how long the benchmark took for some N number of executions. Taking the time difference and dividing by N, one now knows the time per execution of the benchmark code.

```
Static void test_cmd(char *param){
        int t, tf, tfc, i;
        (other declarations needed here)
        t=get_time();
        for(i=0; i<N; i++)result=benchtest(A,B,n);
        tf=get_time();
        tfc=get_time();
        tfc=tfc-tf;  //time it takes for a call to get time
        tf=tf-t-tfc;  //time to execute the benchmark code N times
        //put code here to write the time tf out to the terminal
}
```

Now, add the benchmark code and the test command to the terminal program. Compile and run for both microcontrollers. Report results.

Report your results.

This is an informal report. Include your modified code, showing copies (or well identified excerpts) of anything else you modified. Write a "conclusions" paragraph to say that it worked (or didn't) and remark on anything of interests such as peculiarities, innovative things you did, problems encountered, or things that are still not doing what you want.

A lot of what was done in that 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. Note that this is a different (HCS08) version.

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:\Doccuments and Settings\Administrator\Desktop\EGR298\EGR298init\Sources\derivative.h." Notice the folder "EGR298init" 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.

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.)

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.

A close look at our simple initial C program as it exists in the microcontroller:

C code (comments stripped out):

```
void main(void){
   short i=0;
   PTBDD_PTBDD6=1;
   for(;;){
      __RESET_WATCHDOG();
      PTBD_PTBD6=0;
      i++;
     PTBD_PTBD6=1;
   }
}
```

Disassembled equivalent:

```
E092  AIS   #-2          ;Add immediate value (-2) to stack pointer (to make space for i
E094  TSX                ;Transfer the value in the stack ponter (+1?) to HX
E095  CLR  1,X           ;Clear the byte in RAM pointed to by X+1 (least sign byte of i)
E097  CLR  ,X            ;Clear the byte in RAM pointed to by X (most significant byte of i)
E098  BSET  6,0x03   ;Bit Set at address 0x03 (PTABDD) bit 6 – makes it an output
E09A  STA  0x1800    ;Store whatever is in "A" (it's zero) to address 1800 (feeds dog)
E09D  BCLR  6,0x02 ;Bit Clear at address 0x02 (PTABD) bit 6
E09F  INC  1,X           ;Increment byte pointed to by X+1 (least significant byte of i)
E0A1  BNE  *+3    ; abs= 0xE0A4  Branch to E0A4 if prev. operation didn't give zero.
E0A3  INC ,X             ;Increment byte pointed to by X (most significant byte of i)
E0A4  BSET  6,0x02  ;Bit Set at 0x02 (PTBD) bit 6
E0A6  BRA    *-12    ;abs=0xE09A  Branch always to address E09A
E0A8  BRSET  0,0x00 *-83   ;abs=0xE055  Branch if bit 0 at address 0 is set to E055.
```

Machine language (hexadecimal) in memory:

```
E090    00 00 A7 FE 95 6F 01 7F      AIS   #-2;    TSX;    CLR  1,X;    CLR  ,X
E098    1C 03 C7 18 00 1D 02 6C      BSET  6,0x03; STA  0x1800; BCLR  6,0x02; INC
E0A0    01 26 01 7C 1C 02 20 F2      1,X; BNE  *+3; INC ,X; BSET  6,0x02; BRA *-12
E0A8    00 00 uu uu uu uu uu uu      BRSET  0,0x00 *-83  (we should never get to this)
```

In memory:

Data:1 (globals)

```
_PTBDD   address 0x3
_SRS     address 0x1800
_PTBD    address 0x2
```

Data:2 (locals – on stack)

```
i       address 0x14E
```

A close look at our simple initial C program, with I put in as a global instead of a local variable, +- as it exists in the microcontroller:

C code (comments stripped out):

```
short i=0;
void main(void){
PTBDD_PTBDD6=1;
   for(;;){
      __RESET_WATCHDOG();
      PTBD_PTBD6=0;
      i++;
     PTBD_PTBD6=1;
   }
}
```

Disassembled equivalent:

```
E092  BSET  6,0x03   ;Bit Set at address 0x03 (PTABDD) bit 6 – makes it an output
E094  STA  0x1800    ;Store whatever is in "A" (it's zero) to address 1800 (feeds dog)
E097  BCLR  6,0x02  ;Bit Clear at address 0x02 (PTABD) bit 6
E099  LDHX  #0x0100 ;Load HX with pointer to i (address 0x100)
E09C  INC 1,X          ;Increment byte pointed to by X+1 (least significant byte of i)
E09E  BNE  *+3       ; abs= 0xE0A1  Branch to E0A1 if prev. operation didn't give zero.
E0A0  INC ,X           ;Increment byte pointed to by X (most significant byte of i)
E0A1  BSET  6,0x02  ;Bit Set at 0x02 (PTBD) bit 6
E0A3  BRA    *-15    ;abs=0xE094  Branch always to address E094
E0A8  BRSET  0,0x00 *-86   ;abs=0xE04F  Branch if bit 0 at address 0 is set to E04F.
```

Machine language (hexadecimal) in memory:

```
E090    00 00 1C 03 C7 18 00 1D      BSET  6,0x03; STA  0x1800; BCLR
E098    02 45 01 00 6C 01 26 01      6,0x02; LDHX #0100;   INC1,X; BNE  *+3
E0A0    01 26 01 7C 1C 02 20 F2      INC ,X; BSET  6,0x02; BRA *-12
E0A8    00 00 uu uu uu uu uu uu      BRSET  0,0x00 *-83  (we should never get to this)
```

In memory:

```
Data:1 (globals)
i       address 0x100
_PTBDD   address 0x3
_SRS        address 0x1800
_PTBD      address 0x2
```

```
Data:2 (locals – on stack)
None
```

Code for simple program to light LED to a particular level
In this case, the global variable sets it to “off”.

main.c:

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

short brightness=0; /*brightness, can vary from 0 to 9999*/

void main(void) {
  short j;
  /*EnableInterrupts;*/ /* enable interrupts */
  /* include your code here */
  PTBDD_PTBDD6=1;
  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
      PTBD_PTBD6=0;               /*turns the LED on*/
      for(j=0;j<brightness;j++){}
      PTBD_PTBD6=1;               /*turns the LED off*/
      for(j=brightness;j<10000;j++){}
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

EE342 Microcontrollers
Initial Lab Exercise: Terminal Program

Introduction:

The purpose of this document is to describe the initial exercise basis for projects to be done during the first part of EE342. The idea is to use the PC as a "terminal" (something you can type into and receive and display text) to operate an embedded program on the microcontroller that presents to the user a "command line interface (CLI). This program executes different "commands" that the user types into the terminal program on the PC, and sends back information to be displayed. We use a serial port (which you have already studied in EE247) for the connection between the PC and the microcontroller.

The "terminal" module is "stackware" or "middleware" that came with the DEMOJM boards. It is general in its usefulness, which is what you want from "middleware" that fits between the user application and the device specific drivers that interact with the hardware. For what we are doing, we are using the "terminal" module with a different set of drivers than would be used with the JM family devices.

This document is intended to include information needed to get the terminal program up and running on the S08SH8 processors we will be using. (The same also works with almost any other processor, say, the QG8 from Mechatronics, though perhaps with modification of the drivers.)

Overview:

You will often see diagrams that show the relationship among different modules running in a computer. Usually these are shown with hardware levels at the bottom and the user's application at the top. The applicable diagram for this project is shown in Figure 1. The "terminal" module lies where you would usually find an operating system. Indeed, the terminal can be thought of as a very basic operating system. It receives commands from the user and causes appropriate applications to run, and sends appropriate data back to the user.

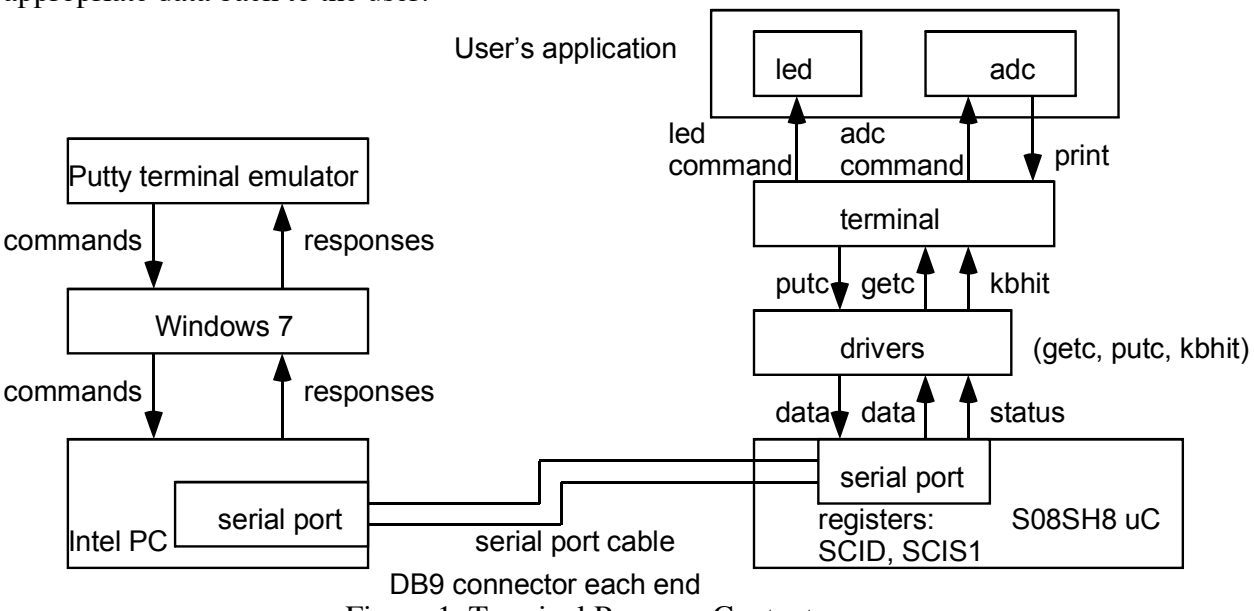

Figure 1 Terminal Program Context

So, for us, we will end up with the following modules that get compiled together to make up the executable for the microcontroller:

1. Main (main.c): main program including initialization, the "main loop" and the application commands (Often we'd separate out initialization and application commands).

2. The Terminal module (hcc_terminal.c, hccterminal.h, utils.c, utils.h) : (You could consider Utils (format conversion utilities) a separate module. We'll lump them in here.)

3. The drivers for the serial port (and related functions): SCI_Functions.c, SCI_functions.h

Other files: We need definitions from mc9s08sh8.h, hcc_types.h, derivative.h for the modules.

Getting it Made:

Code Warrior 10 is picky about its "workspace". It does not like you to copy a project in from somewhere else. It complains. Sometimes it works anyway, but it does complain and there may be some problems down the line. So, we will build this project as if from scratch, but copy in the needed material from files provided (as listed above). So, bring up CodeWarrior 10, and choose where you want your "Workspace" to be. (You could make it a thumb drive, your H drive, or even put it in your user folder on the particular machine's C drive. If you do that, make a copy to a thumb drive, but don't expect CodeWarrior to like it if you try to use it directly.)

Create a new project (I called mine "shterminal") for the S08SH8 processor. You can take the defaults. (You could decide to include floating point; if so make it 32 bit floats.) We will use C, NOT C++. (If you choose C++, a very tight memory restriction is imposed.) Ultimately you end up with a default project with a main and some header files.

Now, add to your project the header files (into the "headers" folder) code files (into the "sources" folder). Be sure they get added to the project, and show up correctly in the project navigator. There will be a default "main". We will want to replace that with the main for this project. You can either replace the file, or just replace the file contents. It should NOT be necessary to move mc9s08sh8.h into the project; that should already be there. It's possible derivative.h will already be there too; I don't remember.

So, with all that done, you should be able to make the project and debug.

Running It:

So, "debug" the (successfully) compiled project. That downloads the code into the microcontroller and sets it up ready to run.

Now, connect the microcontroller to the PC using the serial port cable. Check the strappings on the microcontroller board to be sure that the serial port is enabled.

Next, start PuTTy on the PC. You will get a configuration window. You need to designate use of the "serial" port. You also need to indicate the bit rate: we will use

115200 bits per second. (You sometimes hear this called baud rate.) Once you hit return you should have a blank screen.

It's time to start the microcontroller executing. You should see an initialization message appear on the screen. Then there's a return, and you should see the ">", a prompt symbol, on the left. This is prompting you to give a command. Try typing "help". When you hit return, you should see a response that lists the various commands: help, led, and adc. (Later we'll add more.) try the "led" command. When you hit return, the screen will show another prompt, and an LED on the microcontroller board should flip state between on and off. If you do it again, it flips back.

The "adc" command needs a "parameter". You type "adc 1" top get the potentiometer Voltage, and "adc 2" for the light sensor Voltage. If you type just "adc", the terminal program crashes. You can twiddle the pot or put your hand over the board to shade it to vary the illumination. You get a response to the screen. Is it correct? (It isn't! But, it does vary.)

So, the terminal program is working. It needs some fixing though. For one thing, we'd like correct Voltages. For second, we'd like the adc command not to crash the machine if we forget the parameter. Maybe we'd like to be able to flip either LED. Maybe we'd like an alarm clock that we can set. But to do that, we need to understand how it all works.

The Drivers

Given that you have already studied the serial port, the driver functions should be pretty easily understood. When we initialize the terminal (inside the microcontroller) we need to do several things. We need to be sure we have the microcontroller system clock running at the right speed. We need to initialize the port for the correct bit rate, and turn on the transmitter and receiver. The three functions we need are:

int putch(char), int getch(void), and int kbhit(void).

(You will see these called putc, getc, and various other variations on those names.)

The three functions put a character into the serial port, get a character out of the serial port, and check to see if there is a character waiting to be picked up. An oddity of these functions is that they all return integers (ints), not characters. This is an oddity that goes all the way back to the origins of the C language; it's how these functions have been defined as long as C has been around, since forever. What that means is that we will have to use "casts" to change characters into ints and ints into characters.

Take a looks at all of these functions. Here I will walk through just putch. The function is given below in Figure 2.

```
/*void TERMIO_PutChar(char send)*/
int TERMIO_PutChar(char send){
  int dummy;
  while(!SCIS1_TDRE);/* wait until TDRE=1 */
  dummy = SCIS1;
  SCID  = send;
  dummy=send;
  return dummy; /*added line*/
}//end SPI_PutChar
```

Figure 2 putch function

This is a “synchronous” function. It won’t return until its job is done. (An “asynchronous” function would get something started, then return immediately. The function putch checks to see if the transmit Data Register Empty (TDRE) flag is set. If it is, the data “send” is stuffed into the serial port data register and the function returns the same data “send” that was sent (but as an integer). That’s why the integer “dummy” is used. If TDRE is NOT set, then it just does a spin wait. It could be waiting a looooooong time! At least until whatever character is in the transmit buffer to go out. At 115200 bps that could take as much as 87uS. At 9600 bps (an often-used standard speed) that would take 1.04 mS. That’s a long time.

Notice that these particular function names are TERMIO_PutChar, TERMIO_GetChar, and TERMIO_Kbhit. That’s to distinguish them from other potential putch, getch, and kbhit functions for other kinds of interfaces, say, via USB, which use different registers and code to interact with the device hardware. So, these are the drivers specific to the serial port on an ‘SH8 microcontroller. The terminal module itself is NOT specific to this kind of serial port. In like manner, the initializations are unique to the SH8. That’s why we gather all this SH8 serial port specific stuff into this one “SCI_Functions module (file pair).

The main program needs to call any functions to initialize the serial port before initializing the terminal. So in main, we find the following code fragment as shown in Figure 3 below. After the parallel port to the LED is initialized, the clock and the serial port initialization calls are made. Then the call to initialize the terminal can be made, and here is where we tell the terminal what the functions are to use for putch, getch, and kbhit.

```
PTBDD=0x80;
ICS_FEI_16M();
TERMIO_Init_16M_115200();
terminal_init(TERMIO_PutChar, TERMIO_GetChar, TERMIO_kbhit);
```

Figure 3 SCI Initialization (in main)

Terminal Initialization of Drivers:

Take a look at the file hcc_terminal.c. Near the top you will find several global variables defined. Three of them are shown (an excerpt form the code) in Figure 4 below:

```
static int (*putch)(char);
static int (*getch)(void);
static int (*kbhit)(void);
```

Figure 4 terminal driver function variables

What these three statements do is create three variables which will be known as pointers to functions called (within the terminal module) putch, getch, and kbhit. They are of integer (16 bit) size. When the function terminal_init is called, three pointers are passed, and those pointers are put into these three variables. (The names used in the function declaration have to be different, since the arguments of the function are passed

as stack variables, and we need those values, the pointers to the three functions, to go into global variables.) See Figure 5 below.

```
void terminal_init(int (*putch_)(char), int (*getch_)(void), int(*kbhit_)(void))
{
  cmd_line[sizeof(cmd_line)-1]='\0';
  cmd_line_ndx=0;

  cmds[0]=(void *)&help_cmd;
  n_cmd=1;

  putch=putch_;
  getch=getch_;
  kbhit=kbhit_;

  print_greeting();
  print_prompt();
```

Figure 5 Terminal Initialization function.

Commands and Help:

The terminal works by receiving characters from the user via the serial port, and then doing something appropriate to those commands. So, before we go further, we need to know how a "command" is represented. In terms of C, each command is an instance of a struct called a "command_t". Because commands need to be defined by the user (in other modules) as well as used by the terminal itself, the definition of a "command_t" needs to be "published" by putting it in the file "hcc_terminal.h", not just "hcc_terminal.c". Here's the definition (Figure 6):

```
typedef struct {
  const char *txt;
  cmd_func * func;
  const char *help_txt;
} command_t;
```

Figure 6 Command structure

So, a command has three elements. There is a pointer to a string of text called "*txt". The "const" says that the character data itself is constant; it is stored in FLASH memory (essentially EPROM) and can't be changed. The pointer CAN be changed, because the structure itself is going to be stored in RAM. It's a variable. In fact, there is an array of pointers to these command_t structs called "cmds" (commands). It is declared to exist in the statement:

```
static command_t * cmds[MAX_CMDS];
```

This statement says there is in global variables (and hence static) an array variable "cmds" which consists of some number of pointers to command_t's. So, if MAX_CMDS is defined as 10, there are 10 pointers (of 16 bits each) in this array. Initially the array is empty. There's a global variable n_cmd that tells us how many commands there are, and it starts out undefined. (Actually, yes, this variable is global since any function that knows about it can access it. It is static, meaning it doesn't go away when you are

outside some function; it's persistent. But since it is not in hcc_terminal.h, nobody else knows about it in other modules.)

```
static hcc_u8 n_cmd;
```

So, where are the actual commands that cmds points to? Well, the first one is right here in hcc_terminal.c. It's one of the module variables. Figure 7 shows the "Help" command.

```
static const command_t help_cmd = {
  "help", cmd_help, "Prints help about commands. "
};
```

Figure 7 The Help Command

We saw that the struct for a command_t has three elements. This statement says there is a static global variable called "help_cmd" that is a pointer to a "command_t", the command structure. Furthermore, the code gives a value for this structure (with the "= {….}" part of the statement. The element "txt" will point to a string that reads "help". The element func points to a command function (defined earlier) called "cmd_help". The third variable, "help_txt", points to a string that describes what the help command does. The definition of a cmd_help is earlier:

```
typedef void (cmd_func)(char *params);
```
(from hcc_terminal.h)

So, here again, we have a pointer to a function. This one, generically a "cmd_func" (that's what we call the type of function in this typedef), returns nothing (void) and is passed a string (a pointer to characters) called "params". The name of one of these command functions is "help_cmd", and we can find it right here in the hcc_terminal file (Figure 8). It should be pretty easy to see what this does. It just goes down the list of commands printing the strings it finds.

```
static void cmd_help(char *param)
{
  int x;

  param++;
  print("I understand the following commands:\r\n");

  for(x=0; x < n_cmd; x++)
  {
    print("  ");
    print((char *)cmds[x]->txt);
    print(":\t");
    print((char *)cmds[x]->help_txt);
    print("\r\n");
  }
  print("\r\n");
}
```

Figure 8 Help Command function

You might be wondering about the "params++;" statement. Because this is a command function, there is a pointer that is passed to it. But, the help command does not

need a pointer. But if params was never used in the function, the compiler would generate a warning saying that there is an unused variable. By adding 1 to params we touch the variable just to suppress the warning. (We would not do this if we were low on memory, specifically FLASH memory. But we have 8K. Another byte or two for this line of code won't kill us.)

So, the first thing terminal_init does is install the help command by pointing the first element of cmds to it, then setting the number of commands to 1. Well, not quite the first thing. The first thing it does is initialize the array where the serial port input will be stored, the array "cmd_line". As characters come in, this is where they will go until the terminal recognizes that the user has hit return.

```
cmd_line[sizeof(cmd_line)-1]='\0';
```

Other commands are added by the function: `int terminal_add_cmd(command_t *cmd)`. If you look at the next 2 lines of main after terminal initialization, you will see how the commands "led" and "adc" are added. So, with that, the terminal is ready to run. We enter the main loop.

```
(void)terminal_add_cmd((command_t*)&led_cmd);
(void)terminal_add_cmd((command_t*)&adc_cmd);
```

Terminal Operation:

So, things are set up. What next? As soon as the receiver is turned on the serial port can start receiving characters. But, nothing happens until program execution asks about and picks up those characters. How does that happen? This is a "polled" program. Interrupts are enabled, but there is no interrupt code in the executable code, so the only way something happens is by normal program execution. So, as the main loop executes repeatedly, we find a terminal call:

The main loop is shown in Figure 9 below:

```
for(;;) {
  c=SCIS1;
  terminal_process(); /* services terminal, dispatches commands*/
  DisableInterrupts;
  __RESET_WATCHDOG(); /* feeds the dog */
  EnableInterrupts;
} /* loop forever */
```

Figure 9 main Loop

The reason for the "c=SCIS1" statement is to let the debugger conveniently show the status of the serial port as variable c. It's really not needed. It's the next statement that is key: a call to the function "terminal_process" that actually does the work of the terminal. The function terminal_process checks to see if a character has come in to the serial port. The code for the first part of the function is shown is shown in Figure 10.

```
void terminal_process(void)
{
  char c;    /* moved up one line */
  while(c=(*kbhit)())
  {
    c=(char)(*getch)();
```

```
/*while(c!=(char)(*putch)(c));*/
/* Replace with just a simple call to putch()*/
c=(char)(*putch)(c);
if (c=='\r')
{
  while('\n'!=(char)(*putch)('\n'))
    ;
}
/* Execute command if enter is received, or cmd_line is full. */
if ((c=='\r') || (cmd_line_ndx == sizeof(cmd_line)-2))
{
```

Figure 10 Terminal process and Character pickup

Notice that terminal_process does not communicate with main(); it neither gets arguments nor returns a value. It just happens. The first thing it does is call kbhit to pick up a character called c. Notice the notation for calling a function that is actually pointed to by a variable. It's a bit tricky. (We'll see that again.) There is no calling argument; really all we want to know is whether there's a character to receive. If there is not, we fall out of the while loop and return from terminal_process. If there is, we stay in terminal_process until we run out of characters to pick up. Remember, terminal_process does not know that characters are coming from a serial port one per 84uS at most, and there will never be more than one at a time. There could be a whole buffer full in other circumstances. That's why the while loop; we need to keep doing terminal stuff until we are done; other stuff in the main loop will just have to wait.

If there is a character, we pick it up with getch. Whatever the function was that was passed in during terminal initialization. The character actually comes back as an int, so it must be "cast" as a character for the character variable c, which is where we want to put it. Now, we immediately take the same character and put it back out by calling the putc function.

As the original terminal program had it, the call to putch was in the form:

```
while(c!=(char)(*putch)(c));
```

What that does is keep calling putch until it returns the same value that was passed to it. A non-blocking (asynchronous) version of putch might look and say, "TDRE is 0; we can't send the character out." It would then return a -1 (which is a legitimate int; it's not a legitimate char). Or something else not equal to the character. If that happens, terminal_process turns right around and keeps calling until it gets what it wants, the same character returned as it wanted to send, which indicates that putch has accepted the character and it will eventually go out. Now, in this case putch is doing the same thing; it's checking TDRE to make sure the character can be sent, so it will never return unless it's returning the same value. That's the reason for the simplification, to remove the while loop. (There's a perverse case in which I've seen this thing hang up.)

Next, a test is made to see if the character received is '\r', the "return" character. (we also add a '\n' for new line after the return; otherwise the next line on the terminal will overwrite the line just received.) If the character is not '\r', then we are done; terminal_process returns and the main loop continues.

If a '\r' is received, then terminal process must look down the list of commands it has to see if the first part of the string of characters just receives matches anything. If not, it tells the user that the command is not understood. If a match is found, it calls the corresponding function out of the command_t for that command. When the command function completes, it returns here to terminal_process, which then keeps receiving

characters or returns if no more characters have come in. See if you can follow the logic of the remainder of terminal_process. What you will find is that the parameter passed to the command function, "param", points to the part of the command string just past the command itself. So, if you type "adc 1", then return, the pointer param will point to "1". (Not the numerical value 1, but a string consisting of the asci characters '1' and '\0' , that is 0x31, 0x00.)

User Commands:

The real guts of the user's application consists of the command functions which do the business that needs to get done. In this simple example, there are two of them, "led" and "adc". The "led" command function (from main.c) is given in Figure 11 below.

```
/* command called function to modify LED state */
static void cmd_led(char *param)
{
  static led_on=0;
  param++;
  if (led_on)
  {
    led_on=0;
    PTBD_PTBD7=0;
  }
  else
  {
    led_on=1;
    PTBD_PTBD7=1;
  }
}
```

Figure 11 Toggle the Led Function

Note that here again the parameter "param" is unused. The code just flips the bit at PTBD_PTBD7. (Really good code practice would abstract this into a macro or function.) That's it! There really isn't anything else to do!

The "adc" command is more complicated. It does use param to tell which channel of the A/D converter to collect. See Figure 12.

```
/* function to report ADC results */
static void cmd_adc(char *param){
  static channel=0;
  int i;
  char s[4];
  ADCCFG=0x40; /*10 bit mode*/
  for(i=0;i<10&&param[i]!=0;i++){
    if((param[i]=='1')||(param[i]=='2'))break;
  }
  if(param[i]=='1'){
    /* collect pot value */
    APCTL1=0x01;
    ADCSC1=0;
  } else if(param[i]=='2'){
    /* collect photo value */
    APCTL1=0x02;
    ADCSC1=1;
  }
  while(ADCSC1_COCO!=1);
  i=ADCR;
```

```
  s[0]=((i*5)>>10);
  s[1]='.' ;
  i=i-(s[0]<<10);
  s[0]=s[0]+'0';
  s[2]=((i*50)>>10)+'0';
  s[3]=0;
  print(s);
}
```

Figure 12 Analog data Function

Notice that the basic approach is to look at the parameter passed in from the terminal and see if it is a '1' or a '2' (again, the asci character, not the value). It can be a bit tricky because the user might type in an extra space before the 1 or the 2. Uh oh. Or, what if the user didn't put a '1' or a '2'? Running the A/D converter is straightforward (as done for EGR222). Converting that value to a character string is a challenge. The current code doesn't work! Can you fix it? Hint: the utils.c file contains functions that change integers into strings. But, how do you get an integer for a fractional Voltage? (Second hint: change it to mV then move the decimal). So, see if you can fix this function.

Conclusions:

This document should help you with understanding the terminal program. If you do understand all this, you are doing very well! If not, keep on plugging. Add some more commands. We are going to want to add a clock using one of the device timers. How do you read out the time (from variables containing hours, minutes, and seconds)? How do you set the time? Those will be two commands. Later we will be adding commands to set track Voltages and flip switches. We'll want to be able to monitor track Voltages too. We may want to make receiving characters interrupt driven so that stuff going on in the main loop can't interfere. We may want to buffer outputs and make that interrupt driven as well so that we don't tie up the microcontroller in busy-wait mode while sending long messages with slow characters at 9600 bps. Lots of ways to go from here!

Appendix A Code files:

Headers:

derivative.h:

```
/*
 * Note: This file is recreated by the project wizard whenever the MCU is
 *       changed and should not be edited by hand
 */

/* Include the derivative-specific header file */
#include <MC9S08SH8.h>

#define _Stop asm ( stop; )
  /*!< Macro to enter stop modes, STOPE bit in SOPT1 register must be set prior to
executing this macro */

#define _Wait asm ( wait; )
  /*!< Macro to enter wait mode */
```

hcc_types.h:

```
/*****************************************************************************
 *
 *            Copyright (c) 2006-2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests0
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 *****************************************************************************/
#ifndef _CMX_TYPES_H_
#define _CMX_TYPES_H_

/* Type definitions */
typedef unsigned char hcc_u8;
typedef unsigned int hcc_u16;
typedef unsigned long hcc_u32;

typedef volatile hcc_u8 hcc_reg8;
typedef volatile hcc_u16 hcc_reg16;
typedef volatile hcc_u32 hcc_reg32;

typedef hcc_u8 hcc_imask;

#ifdef NDEBUG
#define CMX_ASSERT(c) (void)0
#else
#define CMX_ASSERT(c)\
do {\
  if(!(c))\
  {\
    int a=1;\
    while(a)\
      ;\
  }\
}while(0)
#endif

#define BREW32(v)   ((hcc_u32)(((hcc_u32)(((hcc_u32)(v)) << 24)) \
                     | ((hcc_u32)(((hcc_u32)(v)) >> 24)) \
                     | (hcc_u32)((hcc_u32)((hcc_u32)(v) & (hcc_u32)0xff00ul) << 8)\
                     | (hcc_u32)((hcc_u32)((hcc_u32)(v) & (hcc_u32)0xff0000ul) >>
8)))

#define BREW16(v)   ((hcc_u16)(((hcc_u16)(v)) << 8) | (hcc_u16)(((hcc_u16)(v)) >>
8))
```

```
#define WR_LE32(a, v) ((*(hcc_u32*)(a))=BREW32(v))
#define WR_LE16(a, v) ((*(hcc_u16*)(a))=BREW16(v))
#define RD_LE32(a)    (BREW32(*((hcc_u32*)(a))))
#define RD_LE16(a)    (BREW16(*((hcc_u16*)(a))))

/* Read 16 bit big endian value from address. */
#define RD_BE16(a) (*(hcc_u16*)(a))
/* Write 16bit value in v to address a in big endian order. */
#define WR_BE16(a, v) (*(hcc_u16*)(a) = (hcc_u16)(v))
/* Read 32 bit little endian value from address. */
#define RD_BE32(a) (*(hcc_u32*)(a))
/* Write 32bit value in v to address a in big endian order. */
#define WR_BE32(a, v) (*(hcc_u32*)(a) = (hcc_u32)(v))

#endif /*_CMX_TYPES_H_*/

/****************************** END OF FILE **********************************/
```

utils.h:

```
/*****************************************************************************
 *
 *            Copyright (c) 2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 ****************************************************************************/
#ifndef _UTILS_H_
#define _UTILS_H_

#include "hcc_types.h"

extern void itoa(int number, char* buf, int length);
extern void itoah(int number, char* buf, int length);
extern hcc_u32 strtoi (char *str);
extern void *_memcpy(void *dst, const void *src, int n);
extern void *_memset(void *s, int c, int n);
#endif
/****************************** END OF FILE **********************************/
```

SCIfunctions.h:

```
#ifndef _SCI_FUNCTIONS_H
#define _SCI_FUNCTIONS_H
```

```
#include "MC9S08SH8.h"

/* added for terminal */
int TERMIO_kbhit(void);

/*void TERMIO_PutChar(char send);*/
/*modified to be compatible with terminal.c */
int TERMIO_PutChar(char send); /*changed from char*/

int TERMIO_GetChar(void);  /*changed from char*/

void TERMIO_Init_20M_115200(void);

void TERMIO_Init_16M_115200(void);

/*added to initiali8ze clock*/
void ICS_FEI_16M(void);

#endif
```

hcc_terminal.h:

```
/*****************************************************************************
*
*            Copyright (c) 2006-2007 by CMX Systems, Inc.
*
* This software is copyrighted by and is the sole property of
* CMX.  All rights, title, ownership, or other interests
* in the software remain the property of CMX.  This
* software may only be used in accordance with the corresponding
* license agreement.  Any unauthorized use, duplication, transmission,
* distribution, or disclosure of this software is expressly forbidden.
*
* This Copyright notice may not be removed or modified without prior
* written consent of CMX.
*
* CMX reserves the right to modify this software without notice.
*
* CMX Systems, Inc.
* 12276 San Jose Blvd. #511
* Jacksonville, FL 32223
* USA
*
* Tel:  (904) 880-1840
* Fax:  (904) 880-1632
* http: www.cmx.com
* email: cmx@cmx.com
*
*****************************************************************************/

#ifndef _TERMINAL_H_
#define _TERMINAL_H_
#ifdef __cplusplus
extern "C" {
#endif

typedef void (cmd_func)(char *params);

typedef struct {
  const char *txt;
  cmd_func * func;
  const char *help_txt;
} command_t;
```

```
extern int terminal_add_cmd(command_t *cmd);
extern int terminal_delete_cmd(command_t *cmd);
extern void terminal_init(int (*putch)(char), int (*getch)(void),
int(*kbhit)(void));
extern void terminal_process(void);
extern int skipp_space(char *cmd_line, int start);
extern int find_word(char *cmd_line, int start);
extern int cmp_str(char *a, char *b);
extern void print(char *s);

#ifdef __cplusplus
}
#endif

#endif

/****************************** END OF FILE **********************************/
```

C Code files:

utils.c:

```
/*****************************************************************************
 *
 *            Copyright (c) 2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 ****************************************************************************/
#include "utils.h"

void itoah(int number, char* buf, int length)
{
  int ndx=length-1;
  int neg;
  buf[ndx--]='\0';

  if (number < 0)
  {
    neg=1;
    number*=-1;
  }
  else
  {
```

```
    neg=0;
  }

  while((number >= 0) && (ndx > 0))
  {
    int digit=number&0xf;
    buf[ndx--]=(char)('0'+digit);
    number>>=4;
  }

  while(ndx > 0)
  {
    buf[ndx--]=' ';
  }

  buf[0]=(char)(neg ? '-' : ' ');
}

void itoa(int number, char* buf, int length)
{
  int ndx=length-1;
  int neg;
  buf[ndx--]='\0';

  if (number < 0)
  {
    neg=1;
    number*=-1;
  }
  else
  {
    neg=0;
  }

  while((number >= 0) && (ndx > 0))
  {
    int digit=number%10;
    buf[ndx--]=(char)('0'+digit);
    number/=10;
  }

  while(ndx > 0)
  {
    buf[ndx--]=' ';
  }

  buf[0]=(char)(neg ? '-' : ' ');
}


hcc_u32 strtoi (char *str)
{
       hcc_u32 rvalue;
  char *c;

       rvalue = 0;

       /* Check for invalid chars in str */
       for ( c = str; *c != '\0'; ++c)
       {
         /* Convert char to num in 0..36 */
         hcc_u8 val=(hcc_u8)(*c-'0');
      rvalue = (rvalue * 10) + val;
       }
```

```
  return rvalue;
}

void *_memcpy(void *dest, const void *src, int n)
{
  int x;
  for(x=0; x<n; x++)
    {
      ((char*)dest)[x]=((char*)src)[x];
    }
    return(dest);
}

void *_memset(void *s, int c, int n)
{
  int x;
  for(x=0;x<n;x++)
  {
    ((unsigned char*)s)[x]=(unsigned char)c;
  }
  return(s);
}

/****************************** END OF FILE **********************************/
```

SCI_Functions.c:

```
#include "SCI_Functions.h"


/* function to see if an incoming character is waiting*/
/* normally this would be in SCI_Functions.c/h but it was missing*/
int TERMIO_kbhit(void){
  int i;
  i= SCIS1_RDRF;
  return i; /*returns 0 if no data,1 if data waiting */
}

/* Internal clock source (ICS) initialization */
/* this is from ICS_Functions.c in the SH8App project*/
/**
 * ICS_FEI_16M: This function sets ICS FEI mode and
   16M BUS clock with trim.
 *
 * Parameters:    void
 *
 * Subfunctions:  none.
 *
 * Return:        void
 */
void ICS_FEI_16M(void)
{
  ICSC1_CLKS  = 0;
  ICSC1_IREFS = 1;
  ICSC1_RDIV  = 0;
  ICSC2_BDIV  = 0;
  ICSTRM      = 0x96;
}


/**
 * SPI_PutChar:   This function sends a character through the SPI.
 *
```

```
 * Parameters:    character to be sent
 *
 * Subfunctions:  none.
 *
 * Return:        void
 */
/*void TERMIO_PutChar(char send)*/
int TERMIO_PutChar(char send){
  int dummy;
  while(!SCIS1_TDRE);/* wait until TDRE=1 */
  dummy = SCIS1;
  SCID  = send;
  dummy=send;
  return dummy; /*added line*/
}//end SPI_PutChar

/**
 * SPI_GetChar:   This function receives a character through the SPI
 *
 * Parameters:    none
 *
 * Subfunctions:  none.
 *
 * Return:        character recieved
 */

int TERMIO_GetChar(void)
{
  char dummy;
  while(!SCIS1_RDRF);/*wait for RDRF=1 */
  dummy = SCIS1;
  dummy = SCID;
  return dummy;
} //end SPI_GetChar

/**
 * TERMIO_Init_20M_115200: This function initial the terminal.
 *
 * Parameters:             void
 *
 * Subfunctions:           none.
 *
 * Return:                 void
 */
void TERMIO_Init_20M_115200(void)
{
  SCIBDH = 0;
  SCIBDL = 0xB;
  SCIC1  = 0;
  SCIC2  = 0x0C;  /*turns on transmitter, receiver*/
}//end TERMIO_Init_20M_115200

/**
 * TERMIO_Init_16M_115200: This function initial the terminal.
 *
 * Parameters:             void
 *
 * Subfunctions:           none.
 *
 * Return:                 void
 */
void TERMIO_Init_16M_115200(void)
{
  SCIBDH = 0;
  SCIBDL = 0x9;
```

```
  SCIC1  = 0;
  SCIC2  = 0x0C;  /*turns on transmitter, receiver*/
}//end TERMIO_Init_16M_115200
```

hcc_terminal.c: (this is the big one)

```
/*****************************************************************************
 *
 *            Copyright (c) 2006-2007 by CMX Systems, Inc.
 *
 * This software is copyrighted by and is the sole property of
 * CMX.  All rights, title, ownership, or other interests
 * in the software remain the property of CMX.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of CMX.
 *
 * CMX reserves the right to modify this software without notice.
 *
 * CMX Systems, Inc.
 * 12276 San Jose Blvd. #511
 * Jacksonville, FL 32223
 * USA
 *
 * Tel:  (904) 880-1840
 * Fax:  (904) 880-1632
 * http: www.cmx.com
 * email: cmx@cmx.com
 *
 *****************************************************************************/
#include "hcc_types.h"
#include "hcc_terminal.h"
#ifdef __cplusplus
extern "C" {
#endif

/*****************************************************************************
 * Macro definitions
 *****************************************************************************/
#define MAX_CMDS    10
/*****************************************************************************
 * Local types.
 *****************************************************************************/
/* none */

/*****************************************************************************
 * External references.
 *****************************************************************************/
/* none */

/*****************************************************************************
 * Function predefinitions.
 *****************************************************************************/
static void print_greeting(void);
static void cmd_help(char *param);
static void print_prompt(void);
static int find_command(char *name);

/*****************************************************************************
 * Module variables.
 *****************************************************************************/
static const command_t help_cmd = {
```

```
  "help", cmd_help, "Prints help about commands. "
};

static char cmd_line[0x20];
static hcc_u8 cmd_line_ndx;

static hcc_u8 n_cmd;
static command_t * cmds[MAX_CMDS];

static int (*putch)(char);
static int (*getch)(void);
static int (*kbhit)(void);
/*****************************************************************************
 * Name:
 *    print
 * In:
 *    s: string
 * Out:
 *    n/a
 *
 * Description:
 *    Print the specified string.
 * Assumptions:
 *
 *****************************************************************************/
void print(char *s)
{
char c; /*added*/
  while(*s)
  {
    /*while(*s != (char)putch(*s))*/
    /* Replace with just a CALL TO PUTCH */
    c=(char)putch(*s)
      ;
    s++;
  }
}

/*****************************************************************************
 * Name:
 *    skipp_space
 * In:
 *    cmd_line: string to parse
 *    start: start at offset
 * Out:
 *    index of first non space character
 *
 * Description:
 *
 * Assumptions:
 *
 *****************************************************************************/
int skipp_space(char *cmd_line, int start)
{
  /* Skip leading whitespace. */
  while(cmd_line[start] == ' ' || cmd_line[start] == '\t')
  {
    start++;
  }
  return(start);
}

/*****************************************************************************
 * Name:
 *    find_word
```

```
 * In:
 *    cmd_line - pointer to string to be processed
 *    start    - start offset of word
 *
 * Out:
 *    Index of end of word.
 *
 * Description:
 *    Will find the end of a word (first space, tab or end of line).
 *
 * Assumptions:
 *    --
 ****************************************************************************/
int find_word(char *cmd_line, int start)
{
  /* Find end of this word. */
  while(cmd_line[start] != ' ' && cmd_line[start] != '\t'
        && cmd_line[start] != '\n' && cmd_line[start] != '\0')
  {
    start++;
  }

  return(start);
}

/****************************************************************************
 * Name:
 *    cmp_str
 * In:
 *    a - pointer to string one
 *    b - pointer to string two
 * Out:
 *    0 - strings differ
 *    1 - strings are the same
 * Description:
 *    Compare two strings.
 *
 * Assumptions:
 *    --
 ****************************************************************************/
int cmp_str(char *a, char *b)
{
  int x=0;
  do
  {
    if (a[x] != b[x])
    {
      return(0);
    }
    x++;
  } while(a[x] != '\0' && b[x] != '\0');

  return(a[x]==b[x] ? 1 : 0);
}

/****************************************************************************
* Name:
*    cmd_help
* In:
*    param - pointer to string containing parameters
*
* Out:
*    N/A
*
* Description:
```

```
*    List supported commands.
*
* Assumptions:
*    --
*****************************************************************************/
static void cmd_help(char *param)
{
  int x;

  param++;
  print("I understand the following commands:\r\n");

  for(x=0; x < n_cmd; x++)
  {
    print("  ");
    print((char *)cmds[x]->txt);
    print(":\t");
    print((char *)cmds[x]->help_txt);
    print("\r\n");
  }
  print("\r\n");
}


/*****************************************************************************
* Name:
*    print_prompt
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Prints the prompt string.
*
* Assumptions:
*    --
*****************************************************************************/
static void print_prompt(void)
{
  print("\r\n>");
}

/*****************************************************************************
* Name:
*    print_greeting
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    --
*
* Assumptions:
*    --
*****************************************************************************/
static void print_greeting(void)
{
  print("This is the simple terminal version 1.0\r\n");
}
```

```
/*****************************************************************************
* Name:
*    find_command
* In:
*    name - pointer to command name string
*
* Out:
*    number - Index of command in "commands" array.
*    -1     - Command not found.
*
* Description:
*    Find a command by its name.
*
* Assumptions:
*    --
*****************************************************************************/
static int find_command(char *name)
{
  int x;
  for(x=0; x < n_cmd; x++)
  { /* If command found, execute it. */
    if (cmp_str(name, (char *) cmds[x]->txt))
    {
      return(x);
    }
  }
  return(-1);
}

/*****************************************************************************
* Name:
*    terminal_init
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Inicialise the terminal. Set local varaibles to a default value, print
*    greeting message and prompt.
*
* Assumptions:
*    --
*****************************************************************************/
void terminal_init(int (*putch_)(char), int (*getch_)(void), int(*kbhit_)(void))
{
  cmd_line[sizeof(cmd_line)-1]='\0';
  cmd_line_ndx=0;

  cmds[0]=(void *)&help_cmd;
  n_cmd=1;

  putch=putch_;
  getch=getch_;
  kbhit=kbhit_;

  print_greeting();
  print_prompt();
}

/*****************************************************************************
* Name:
*    terminal_proces
* In:
```

```
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Main loop of terminal application. gathers input, and executes commands.
*
* Assumptions:
*    --
*******************************************************************************/
void terminal_process(void)
{
  char c;    /* moved up one line */
  while(c=(*kbhit)())
  {
    c=(char)(*getch)();
    /*while(c!=(char)(*putch)(c));*/
    /* Replace with just a simple call to putch()*/
    c=(char)(*putch)(c);
    if (c=='\r')
    {
      while('\n'!=(char)(*putch)('\n'))
        ;
    }
    /* Execute command if enter is received, or cmd_line is full. */
    if ((c=='\r') || (cmd_line_ndx == sizeof(cmd_line)-2))
    {
      int start=skipp_space(cmd_line, 0);
      int end=find_word(cmd_line, start);
      int x;

      /* Separate command string from parameters, and close
         parameters string. */
      cmd_line[end]=cmd_line[cmd_line_ndx]='\0';

      /* Identify command. */
      x=find_command(cmd_line+start);
      /* Command not found. */
      if (x == -1)
      {
        print("Unknown command!\r\n");
      }
      else
      {
        (*cmds[x]->func)(cmd_line+end+1);
      }
      cmd_line_ndx=0;
      print_prompt();
    }
    else
    { /* Put character to cmd_line. */
      if (c=='\b')
      {
        if (cmd_line_ndx > 0)
        {
          cmd_line[cmd_line_ndx]='\0';
          cmd_line_ndx--;
        }
      }
      else
      {
        cmd_line[cmd_line_ndx++]=c;
      }
    }
```

```
  }
}

/*****************************************************************************
* Name:
*    terminal_add_cmd
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Main loop of terminal application. gathers input, and executes commands.
*
* Assumptions:
*    --
*****************************************************************************/
int terminal_add_cmd(command_t *cmd)
{
  if (n_cmd >= MAX_CMDS)
  {
    return(1);
  }

  cmds[n_cmd]=cmd;
  n_cmd++;
  return(0);
}

/*****************************************************************************
* Name:
*    terminal_delete_cmd
* In:
*    N/A
*
* Out:
*    N/A
*
* Description:
*    Main loop of terminal application. gathers input, and executes commands.
*
* Assumptions:
*    --
*****************************************************************************/
int terminal_delete_cmd(command_t *cmd)
{
  int x;

  for(x=0; x<n_cmd; x++)
  {
    if (cmds[x] == cmd)
    {
      while(x<n_cmd-1)
      {
        cmds[x]=cmds[x+1];
        x++;
      }

      cmds[x]=0;
      n_cmd--;

      return(0);
    }
  }
```

```
  return(1);
}

#ifdef __cplusplus
}
#endif
/****************************** END OF FILE **********************************/
```

main.c:

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

/* command called function to modify LED state */
static void cmd_led(char *param)
{
  static led_on=0;
  param++;
  if (led_on)
  {
    led_on=0;
    PTBD_PTBD7=0;
  }
  else
  {
    led_on=1;
    PTBD_PTBD7=1;
  }

}

/* the command to modify the LED state */
static const command_t led_cmd = {
  "led", cmd_led, "Toggles leds state."
};

/* function to report ADC results */
static void cmd_adc(char *param){
  static channel=0;
  int i;
  char s[4];
  ADCCFG=0x40; /*10 bit mode*/
  for(i=0;i<10&&param[i]!=0;i++){
    if((param[i]=='1')||(param[i]=='2'))break;
  }
  if(param[i]=='1'){
    /* collect pot value */
    APCTL1=0x01;
    ADCSC1=0;
  } else if(param[i]=='2'){
    /* collect photo value */
    APCTL1=0x02;
    ADCSC1=1;
  }
  while(ADCSC1_COCO!=1);
  i=ADCR;
  s[0]=((i*5)>>10);
  s[1]='.' ;
  i=i-(s[0]<<10);
  s[0]=s[0]+'0';
  s[2]=((i*50)>>10)+'0';
  s[3]=0;
```

```
  print(s);
}

/* command for ADC status */
static const command_t adc_cmd = {
  "adc", cmd_adc, "Gives A/D result: 1 for pot / 2 for photo"
};

/* Main program */
void main(void) {
  char c;

  EnableInterrupts; /* enable interrupts */
  /* include your code here */
  PTBDD=0x80;
  ICS_FEI_16M();
  TERMIO_Init_16M_115200();
  terminal_init(TERMIO_PutChar, TERMIO_GetChar, TERMIO_kbhit);
  (void)terminal_add_cmd((command_t*)&led_cmd);
  (void)terminal_add_cmd((command_t*)&adc_cmd);
  /* main loop */
  for(;;) {
    c=SCIS1;
    terminal_process(); /* services terminal, dispatches commands*/
    DisableInterrupts;
    __RESET_WATCHDOG(); /* feeds the dog */
    EnableInterrupts;
  } /* loop forever */
  /* please make sure that you never leave main */
}
```