EE 342 Microcomputer Based System Design
Spring, 2017

Texts:
Bannoura, Bettelheim and Soja, *Flexis and Coldfire V1 Microcontrollers*, AMT Publishing, ISBN 0-9762973-2-9.
Ivor Horton, *Beginning Visual C++ 2008*, Wrox / Wiley Publishing, 2008, ISBN 978-0-470-22590-5 or Horton, *Beginning Visual C++ 2012*, 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 HCS08JM series microcontroller documentation, available from www.nxp.com, (formerly Freescale)and similar materials for the Coldfire V1.
Useful references:
Fabio Pereira*, HCS08 Unleashed, Designer's Guide to the HCS08 Microcontrollers*, 2nd ed., 2009, ISBN 1-4196-8592-9
Jan Axelson, *USB Complete*, 4th ed., Lakeview Research, 2009, ISBN13 978-1-931448-08-6.

Scheduled class times: MW 10-11:45 SLC224
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) HCS08JM60 and Coldfire V1. The HCS device is a relative of the HCS08QG8 and HCS08SH8 used in the Mechatronics and Embedded computers classes. The HCS08JM60 includes a microprocessor core, memory, and a variety of interfaces, including parallel ports, serial ports, A/D converter, and (unlike the QG8 or SH8) a USB port. We will shift to using the MCF51JM128, a "Coldfire" V1 16/32 bit microcontroller derived from the earlier M68000 microprocessor early in this course. We will develop programs to run on the microcontroller using the "Codewarrior" development software. The CD-ROM version with the demo package isn't usable. For Windows 7 or later, you'll have 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, an accelerometer, the USB port hardware, and debugging support. (The software and documentation can be downloaded directly from the Freescale web site. The product we are using is identified as the "DEMOJM" board. You'll get this at the first class.)

Ultimately, 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.

Several years ago and earlier, EE342 projects using the (Motorola Semiconductor, which became Freescale, now NXP) M68000 family (predecessor of the Coldfire V1) focused more on hardware, and did not attempt to communicate with anything more sophisticated than a classical RS232 serial port. You don't find simple serial ports so much anymore; everything seems to have gone to USB (or other) interfaces. This newer version of the course 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 Jan 18*	Overview and Introduction	DEMOJM "labs"
2 Jan 23	Programming basics	Horton 1-6, handout
2 Jan 30	Programming basics (continued) HCS08 and Coldfire	Bannoura intro chapts
3 Feb 6	Codewarrior and programming, debugging	NXP/Freescale stuff
4 Feb 13	More programming, basic interrupts and I/O	DEMOJMstuff, U5,6
5 Feb 20	Coldfire MFC51JM128: the core and capabilities	Bannoura ch…
6 Feb 27	HCS08 and Coldfire I/O ports	Bannoura ch…
7 Mar 13	Sensing things, controlling things	NXP stuff test#1
8 Mar 20	Communications basics: use USB as serial port	DEMOJM stuff
9 Mar 27	Interrupts, timing	Bannoura, NXP docs
10 Apr 3	USB ports	Bannoura, USB book
11 Apr 10	Visual Studio, Applications intro	Horton,handout test #2
12 Apr 17	Host software	Horton
14 Apr 24	Host software, project demonstration (to be scheduled)	Horton
15 May 1*	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. As such, the two books (Bannoura and Horton) should not be terribly expensive. We will use selected parts, but mostly they are there so that when you need to know something, you can dive into the appropriate book to find it. We will use Bannoura et. al. throughout the course. It's a good description of the microcontrollers and what it can do. It also 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 Freescale and elsewhere, but it's nicely collected and presented. It replaces the Pereira book from EE247 because it includes the Coldfire V1 as well as the HCS08 family, which we will use most of the semester.

The Horton book is on our list for two reasons. First, it has good introductory programming stuff for using C and C++. Second, it has help for using Visual Studio 2008, 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. You can use Visual Studio 2008; both are on the computers in the lab. 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. You may be able to get by fine without this book 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/Freescale. Many of these reference documents can also be found on the CD that comes with the microcontroller. In particular, we will need that as a source of detailed information on both processors. It would be nice to have additional reference books, but this pair seemed the right place to stop.

Each student will receive a "DEMOJM" board that includes the HCS08JM60 and MFC51JM128 (Coldfire V1) microcontrollers and hardware support for programming, debugging, and an assortment of devices for demonstration purposes. With the board you also get some documentation and a CD. The documentation includes a series of "labs", that is, exercises, that you can do to get a sense of what you can do with the microcontroller and how you can do it. For example, the first demo has the microcontroller emulate a computer "mouse." It causes the cursor to move back and forth on the screen. Another allows you to type in commands on the PC to flip the LED's or monitor the pushbuttons. Yet another "lab" allows you to read switches and flip LED's using a host application, clicking on icons with your mouse on the PC screen. You should start the course by stepping through these demonstrations. That's our first lab exercise. They are pretty straightforward and should demonstrate the ideas behind a lot of what we will be doing. (To look at this in advance, go to the NXP/Freescale web site and look for the DEMOJM.)

Lab schedule: We will start with the DEMOJM 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 NXP/Freescale "lab" exercises
2. Text based interaction, using peripherals (A/D, PWM)
3. Interacting with the real world: sensors and actuation
4. Interacting with the user via USB and your PC application
5. Performance assessment (benchmarking)

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 SLC224, but it will be the first time with the existing lab arrangements. 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 happening in the lab. With only a small crew this offering, we shouldn't have too much trouble doing this, but may need to move sonme stations around. 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%
Final exam: 25%
Final Project and Report: 20%
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/Freescale web site. Even then, there are issues. I have some documentation that helps with adapting to Codewarrior 10. Codewarrior 10.x interfaces look a bit different from those of 6.x, but Codewarrior 10 is better, especially for the JM series microcontrollers, since it can give you a debug print (serial) port.

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.

EE342 Microcomputer Operation and Design
Laboratory Exercise #1

Jan 25, Feb 1; Report due Feb 8

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 DEMOJM board.

Preliminaries:

Install the software that comes with the microcontroller in a computer you can use without having to worry about being an "administrator" in order to hook up new USB devices or compile and run executables. (The SLC224 computers should be fine. The SLC216 computers won't be fine. The lounge computers are an unknown.)

Run through the series of "Labs." Be sure to at least get through the cdc_terminal_demo exercise, since we will be using that code as a departure point for this lab exercise.

Read up on the HCS08 and Coldfire V1 a bit, and the DEMOJM board. In particular, figure out what ports and bits of those ports go to the various LED's and pushbuttons. (The schematic is a big help for that.) Look through the various source and header files of the cdc_terminal_demo project in CodeWarrior, and get used to how things are organized and accessed.

Try the introductory lab exercise from EE247 on your own (see reference), but using the JM60 instead of the SH8 processor.

Procedure:

(This is adapted from earlier CW 6.x material; CW 10 may be a bit different in places.)

1. Get your files set up and ready

First, create a new folder and project simply by copying the existing cdc-terminal-demo folder. The idea is to leave all of the original stuff copied from the CD untouched. Then, rename the folder to something you wan to call the project, and rename the project (".mcp" file) inside the folder with the same name. (I suggest you turn on extensions for all of the files in your projects when viewing folders.) Now you have your project. If you double click on the renamed .mcp file, it should open up in CodeWarrior and look just like the original cdc-demo.mcp project. (Try making it and running debug just as in the intro "lab" to ensure that it still works.)

Recall that we want to leave original files untouched. Here's the problem: a lot of the files in this project, most of them in fact, are not in the "Sources" folder of the project. (Only two of them are: derivative.h and Start08.c! Not even main!) The "project" view in CodeWarrior does NOT correspond to where the files are in the directory structure. The other source files are in folders that are "common" to several of the demos. For example, "usb.h" and "usb.c" are used by all of the demos. If we modify them, we could mess up the other demos. (I've done that!) Take a look in the Sources folder. What we want to do is copy any file we need into that Sources folder, then substitute it for the file we are connected to now. There are several ways this can be done. Here's what I suggest.

For each file in 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 hyperterminal as in the introductory lab.

Plan B: If you have problems, you can just skip all this above and modify things wherever they are. It will goof up the code for later projects and lab work, but you can always copy it all again from the CD! That's essentially what I did the first time, and decided to keep all the files local the next time. I think switching to local copies is a worthwhile exercise because you get familiar with fooling with files in projects. It also means that you can take your WHOLE project with you just by copying its folder onto a thumb drive. That will be very handy. It also allows making a backup to be easier. It's a good idea to make a backup copy before you start to make a series of modifications, say, the rest of the stuff you are going to do for this lab exercise.

2. Modify the code to control all of the LED's, and to monitor 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 4" you flip the 4th 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 Hyperterminal 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 JM16 (or QG8, etc.), 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 the JM60.) 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 cdc_process();. Run it and see it work!

3. Do the Coldfire processor exercises similarly.

There is a separate set of folders for the CMX stuff for the MFC51JM128. They appear when you do the Coldfire installation. Some of the exercises are the same; you can skip those (mouse, keyboard etc.). But the two serial ports exercise and especially the thumb drive exercise are well worth doing. Notice the differences between the Coldfire assembly language and the HCS08 assembly language. We won't try to modify the Coldfire code now, we just want to see it in action.

4. Report your results.

This is an informal report. Include your modified code from target.c, a copy of main.c, and copies of anything else you modified. Also, include a transcript of a session in Hyperterminal 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.

5. References:

Two documents from EE247 may be helpful. Next time EE342 is offered, EE247 will be a prerequisite. These exercises were done with the HSC08QG8 (the microcontroller from Mechatronics) but similar things can be done with the JM60, your microcontroller. These should be helpful in understanding what is going on. Copies are appended (at the end of the syllabus package).

EE247: Programming for Embedded Systems Lab demo Jan 19, 2012
EE247 Code example

EE342 Microcomputer Operation and Design
Laboratory Exercise #2

February 8-24; Report due Mar 1

Objective:

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

Preliminaries:

Read up on the HCS08 interrupt. Clock/timer. And A/D material. Study the DEMOJM 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 simply copying the folder for the previous project, and renaming the folder and the 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 accelerometer

1) Figure out how to initialize the various signals and ports to allow reading the 4 signals of interest with the A/D converter. Put your code to do that into target.c. You may want some other stuff in target.h too.
2) Add commands to read the potentiomenter, and to read the accelerometer. In response to the command, print the Voltage to the screen. Use 12 bits of resolution.

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.

5. Make it work on the Coldfire V1.

Make all this with the Coldfire, or (acceptable) even do it all with Coldfire from the beginning. The main difference is that you have different interrupt numbers. The point here is that we want to be shifted to doing our lab work with the Coldfire processor from this point on.

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 in Hyperterminal showing a "help" command, then several commands set and read the clock, read the pot at a couple of different values, and read the accelerometer x,y,z for a couple of different positions. Also include in the Hypertemninal 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

March 1,15; Report Mar 22

Objective:

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

Preliminaries:

You might want to create yet a new project for this lab. I suggest simply copying the folder for the previous project, and renaming the folder and the project.

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.) 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 4 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 USB / 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 Hyperterminal 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

March 22,29; report April 5

Objective:

Introduction to programming under Windows and communications via USB to the microcontroller. Do this exercise with Coldfire.

Preliminaries:

Copy the "PC side" Visual Studio project for the hid demo to the Visual Studio 2008 (or 2012) 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. 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

April 5 (start); report May 3 (after end of semester)

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 <u>in discussion</u> 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 */
}
```