EE 342 Microcomputer Based System Design Spring, 2020

Potentially Useful Texts (not required):

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 "Kinetis" series microcontroller documentation, available from www.nxp.com, (formerly Freescale).

Scheduled class times: TBD SLC238 (SLC193?) Likely TR 6PM, subject to change
Instructor: John B. Gilmer Jr. Office hours: TBD Office:SLC220 Phone: x4885
Pre/co requisites: EE241 Digital Design, plus EE247, CS126, CS246 or equivalent

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 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 we have EE247/CS126/CS246 as 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) "Kinetis" family KL43Z. 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. The microcontroller comes on a small inexpensive PC board that includes a variety of interfaces and devices, including two LED's, pushbuttons, a pot an accelerometer (sensor), an LCD display, a USB port, and debugging support. (The software and documentation can be downloaded from NXP.)

Ultimately, we will use the USB interface. Using USB is considerably more complicated than serial ports. 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.

Week of:	Topics covered	Reading, Tests
1 Jan 13	Overview and Introduction	Terminal code, IDE
2 Jan 20*	Programming basics, parallel ports	Terminal, IDE
2 Jan 27	More programming, basic interrupts and more I/O	uP man., examples
3 Feb 3	Structures, Processes for port pins	uP man. Report#1
4 Feb 10	Communications basics, serial port, terminal	
5 Feb17	Timers, clocks, and Pulse Width Modulation	test #1 (ports)
6 Feb 24	Buffering, timing issues, priorities	handouts. Report #2
7 Mar 9	Sensing things, controlling things	
8 Mar 16	Power handling, static, noise, and Voltage hazards	
9 Mar 23	USB ports, Interrupts, timing with USB	Report #3
10 Mar 30	Visual Studio, Applications intro	Horton, tutorials
11 Apr 6*	Visual Studio, Applications (continued)	Horton, tutorials
12 Apr 13	Host software	test #2 (electr, control)
14 Apr 20	Host software, project demonstration (to be scheduled)	Horton
15 Apr 27*	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 book mentioned (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 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 later versions of Visual Studio are out as well. I believe both 2012 and 2015, maybe later, are in the labs. The earlier version of Horton (for 2008) ought to be available cheaper. As a student, you should be able to get Visual Studio for free. 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.

Each student will receive a "Demo" board for the KH43Z 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 emulation program (e.g. PuTTY) on the PC. The terminal program can execute 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. We'll start doing this very soon in the course.

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

About the lab Exercises:

This EE342 offering is transitional. Until recently we used the freescale / NXP "Coldfire" processor with Code warrior for the IDE. That processor was discontinued. In the last offering, a transition was made toi using the KL43Z, but not everything was successfully resolved, in particular the USB uC-PC link. That has now been successfully

demonstrated, but this is the first time we are doing that with the KL43Z as part of the course. There may yet be some surprises and discoveries. If so, we will deal with those as we meet them. The lab descriptions, especially for Labs #1, 2, and 3 may need modification. We will see.

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 or SLC193 or perhaps elsewhere. We need a spot where we can leave things set up, and for now that is TBD, 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 5 students this offering, we will have to configure the track in a linear shape that will need about 12 to 14 feet of space. We will want to determine who has what track section 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, 4 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%	Final Project and Report: 21%
Four lab exercises at 6% each: 24%	Pop quiz(zes): 2%
Final exam: 25%	Class participation: 3%

Windows 7/10 and IDE:

The SLC238 lab computers currently run Windows 7. Those in SLC193 run Windows 10. We will be using MCUXpresso the IDE (Integrated Development Environment) available through the NXP support page (on the NXP web site) for the KL43Z processor and development board. Some of the materials developed in the passed used KDS (Kinetis Development System) but there does not seem to be much difference in how they work. Getting used to these new tools and the kinetix family devices is a work in progress, though, and we can expect to run into issues of one sort or another as we go.

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 Jan 27; Report due Feb 3

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 KL43Z, 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, and the supplied documentation, and get used to how things are organized and accessed.

Get the terminal program working. We will make sure everybody gets to this point in lab, probably the first week. Also, refer to the tutorial document "Getting started with the FRDM-KL43Z Demo Board and MCUXpresso." Walk through the exercise there.

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.)

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 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 both. (Be sure you understand what's going on here.) Now look at the code for the pushbuttons. You want to be

able to read both. 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 both 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 2 variables to save the states.) 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.
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 Feb 17; Report due Feb 24

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.

Procedure:

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

1. Add command to read the accelerometer.

1) Figure out how to initialize the various signals and ports to allow reading the 3 signals (one per axis) of interest with the A/D converter.
2) Add a command to read the accelerometer. In response to the command, print the Voltages to the screen. Use 12 bits of resolution.

2. Add a clock to your project (RTC Timer)

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){  /** This is from SH8 – different for KZ43Z**/
```

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 Mar 16; Report Mar 23

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:

See the tutorial document "Generic HID USB Interface for the NXP FRDM-KL43Z Demo Board for EE342". Follow that to get the Windows PC to be able to flip the two LED's and monitor the two 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. On the uC: Report back to PC an A/D Voltage (externally set to some pin by a potentiometer)
2. Display on the screen the A/D Voltage read.

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 almost the same USB stuff. (You need to provide for getting the data to the PC.) 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 A/D code to read the external pot. (I suggest testing that using the terminal program just to make sure it works.) This means first going into the microcontroller code and providing similar code for the accelerometer, which is pretty straightforward.

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 uC may be big endian, the PC is Intel, hence little endian. The string displayed in the CEdit may be Unicode rather than ascii; if so 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!)

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)

Additional materials TBD later.