EE 472 Lab 1

Introducing the Lab Environment

University of Washington - Department of Electrical Engineering

Blake Hannaford, Shwetak Patel, Mitch Ishihara, Josh Fromm, Brad Emerson, Anh Nguyen, Greg Lee, Jered Aasheim, Tom Cornelius, James K. Peckol

Introduction:

This project has two main purposes. The first is to introduce the C language, explore some of the important aspects of the language including pointers and multiple file programs, then practice simple debugging. The second is to introduce the Texas Instruments Stellaris EKI-LM3S8962 (gees, that's a lot of numbers and letters) implementation of the ARM Cortex-M3 ARMv7-M microcomputer, the IAR Systems Embedded Workbench integrated C / Assembler development environment, and the world of embedded applications. Wow, so, when is the final project due? Like a typical embedded system, we have a hardware piece and a software piece. This term, our major focus will be on the software piece.

You are strongly encouraged to read the entire project before trying to start designing or exploring. The best way to become familiar with any new piece of hardware is to take a tour of its features; the same holds true for learning a new piece of software or programming language. This is the approach that we will take.

We will take our first steps with the C language by starting with a working C program. We will make several modifications to the program then compile and run our versions. Next, we will work with several programs that compile and appear to work properly and yet contain several common errors. Using our debugger in the IAR IDE, we will try to identify and correct those errors. Then, we will build our project, compile it, download it, and run it on the target platform.

We will take the same approach as we move on to more complex projects in future. As we explore the processor, we'll begin with the data sheet and discover some of the key features of this device. This is exactly how we will do it in future when we start to work with a new microprocessor for our company or in our graduate research.….then, on the second day…

Prerequisites:

Familiarity with data types, data structures, as well as standard program design, development, and debugging techniques. No beer until the project is completed. I really mean it.

Background

For us, this is a new microprocessor and development environment. Learning new components and tools is exciting and challenging. This is one of the fun parts of engineering. At the same time, sometimes it can be frustrating when the hardware or the software doesn't behave the way we want it to, or when, unfortunately, it behaves exactly the way that we have told it to, or when we can't immediately find the answer to our questions and problems.

At this stage in learning engineering and about the engineering process, playing with the toys often seems to be the most exciting part…the documentation, formal design, and so on, is, well, often seen as rather boring. *Why do I have to go through all this…why can't I just go to the Internet and find*

something like the design and make a few modifications and be done? In the real-world, the documentation and formal design are absolutely critical parts of all phases of the engineering development process. Doing it and doing it right can mean the difference between success or failure of a project, the malfunction of the system following delivery to the customer, or the possible loss of life while it is being used.

It's also very important to recognize and to remember that the answers to most interesting real-world engineering problems originate in our brains, discovered as we use our imaginations and knowledge to creatively apply the underlying theory; they are reached through our persistent hard work and diligence. The solutions to challenging problems are not sitting, ready, and waiting for us on the Internet. The Vikings didn't discover North America by searching some ancient version of the Internet…they took risks…they explored and challenged. We didn't find the solution for making the first successful airplane flight, to putting someone on the moon, to making the first soft landing on Mars, or to designing and programming the first microprocessors, or the discovery of the Higgs boson on the Internet…we won't find the answers to many of today's problems or the projects this quarter there either. We challenge you to explore, to think, and to make those discoveries.

Sometimes your instructor or TAs will have the answers and sometimes we won't. As we said in the opening, bear in mind that we're (starting to work with) learning the environment and tools too. This platform and development environment are complex and are new to all of us (but we're learning). If we all work together, to identify and solve problems as they occur, everyone benefits and we help to build for the next classes. The answers are there somewhere. Let's all work to find them.

Regardless of the specific platform/environment used, embedded systems development requires at least the following:

1. An understanding of the problem that we are trying to solve.
2. A target platform on which to develop the application.
3. A mechanism for programming the target platform.

Target platform is a term used loosely in industry to mean the actual piece of hardware that performs the computation and/or control – the place where our application (the software) will ultimately reside and run.

A target platform can be as simple as a single chip microprocessor (i.e. a PIC or an Arduino), or as complicated as a feature-rich single board computer (build around a multicore high performance processor, several special purpose processors, and possibly several programmable logic devices). Despite the differences in physical characteristics, the target platform's purpose is the same: to execute the software written by the developer. Here, and for the typical microprocessor based application, the software will be written in the C language. On many occasions, the C++ language may be used; for what are called hard real-time systems, Java is rarely used. On a programmable logic device like an FPGA, it may be a mixture of C and Verilog, for example.

The term *to program* here has two meanings. The first is the more traditional embedded sense and simply means writing software to control a given piece of hardware (in this case the target platform and any peripheral devices that may be connected to it – see above). In the embedded world, *programming* the target also means storing the executable into memory on the target. As the characteristics of the target platform can vary greatly, so too can the mechanism used to program the target platform.

Some development environments allow one to develop directly on the target platform (much like developing code on a PC), while others require that code be developed on another *host computer* and

then transferred to the target platform. We call this transfer *downloading to the target platform*. Apart from the actual technique(s) used, every embedded system has to provide a mechanism for programming it.

One of the more difficult concepts to learn in C is the notion of pointers. Once you begin to see what they are and how they work, you'll find that they're actually rather straightforward. In reality, pointers are nothing more complicated than a variable type whose *value* is *interpreted* as the *address* of something in memory. This is exactly the same as interpreting a collection of bits (they're just bits) as an integer, a character, a floating point number, some interesting picture that you downloaded from one of those special websites on the internet (don't download stuff here using BitTorrent – they have no sense of humor and will shut the university's network down), or all the data on your hard drive as a really really big integer. Weird, huh?

There is a very good explanation of pointers with accompanying drawings in your text. Once again, take the time to read through this material.

Development Environment

The development environment will be the Stellaris system. Sorry, no drones this quarter again…how about an R/C car? This system is a feature-rich single board computer equipped with an ARMv7-M architecture Cortex-M3 microprocessor which is a follow-on from the ARMv4 architecture ARM7TDMI microprocessor.

As a point of interest, among the ARM's early ancestors was the BBC Microcomputer from Acorn Computers, Ltd. The official *Acorn RISC machine* project was started in October 1983.

From Wikipedia…

> *Acorn Computers was a British computer company established in Cambridge, England, in 1978. The company produced a number of computers which were especially popular in the UK. These included the Acorn Electron, the BBC Micro and the Acorn Archimedes. Acorn's BBC Micro computer dominated the UK educational computer market during the 1980s and early 1990s, drawing many comparisons with Apple in the U.S.*

Although the company was broken up into several independent operations in 1998, its legacy includes the development of RISC personal computers. A number of Acorn's former subsidiaries live on today – notably ARM Holdings who are globally dominant in the mobile phone and PDA microprocessor market. Acorn is sometimes known as "the British Apple".

Interestingly, the ARM family of microprocessors are actually a design (piece of intellectual property) licensed by ARM to a variety of different companies to build.

Does anyone in industry really use these things? Well, let's see. By 2005, about 98 percent of the more than one billion mobile phones sold each year used at least one ARM processor. Two years ago, ARM processors accounted for approximately 90% of all embedded 32-bit RISC processors.

The Cortex-M3 is a RISC – Reduced Instruction Set Computer – processor that runs at up to 50 MHz yet, we observe that all CPU hertzes are not created equal. Based upon its architecture and the way that the machine operated, a Cortex-M3 running at 50 MHz has the potential of being more powerful than a Pentium running at the same speed.

The ARMv4 architecture ARM7TDMI supports two instruction sets: a 32-bit implementation (called ARM) in support of high performance embedded designs and a 16-bit implementation (called Thumb – upgraded to Thumb-2 in the Cortex-M3) that supports a reduced memory footprint for

smaller embedded applications. Like the ARM7TDMI, the Cortex-M3 utilizes a three-stage pipeline architecture: instruction fetch, decode, and execute. We'll talk about what all these mean in class.

The version of the processor on the Texas Instruments Stellaris is the ARM Cortex-M3 based on the ARMv7-M architecture.

Some other features include:

- Memory
  - 64 KB SRAM
  - 256 KB Flash memory
- Interrupt Controller and programmable number of external interrupt inputs
- Hardware-division and single-cycle-multiplication
- I/O Communication
  - Seven General Purpose I/O blocks – 5-42 programmable I/O pins
  - 2 UARTs with IrDA support
  - I^2C, SSI and CAN interfaces
  - USB port
  - Ethernet interface
- JTAG Boundary Scan interface
- Debug Support
- Internal Peripherals
  - 4 32-bit timers
  - Real time timer
  - 6 Channel PWM Controller
  - 4 Channels 10-bit A/D
  - Analog Comparator

In this class, we will use many of these hardware features as we design and develop the various design projects; the details regarding each of the components will be given as needed.

To program the Stellaris Cortex-M3 target platform, a host PC is required. Using the *IAR Systems Embedded Workbench*, code can be developed on a standard PC, cross compiled for the Cortex-M3, and then downloaded into the target platform over a USB serial connection.

Project Objectives

- Introduce the C language, C programs, and the PC development environment: the basic C/C++ preprocessor, program structure, multiple file programs, pointers, passing and returning variables by value and by reference to and from subroutines, designing, compiling, and debugging on the target platform.
- Introduce the working environment: the embedded development environment, the target platform, host PC, equipment, etc.

- Learn a bit about the Texas Instruments implementation of the Cortex-M3 processor.

Learning the Environment and Tools – The First Steps

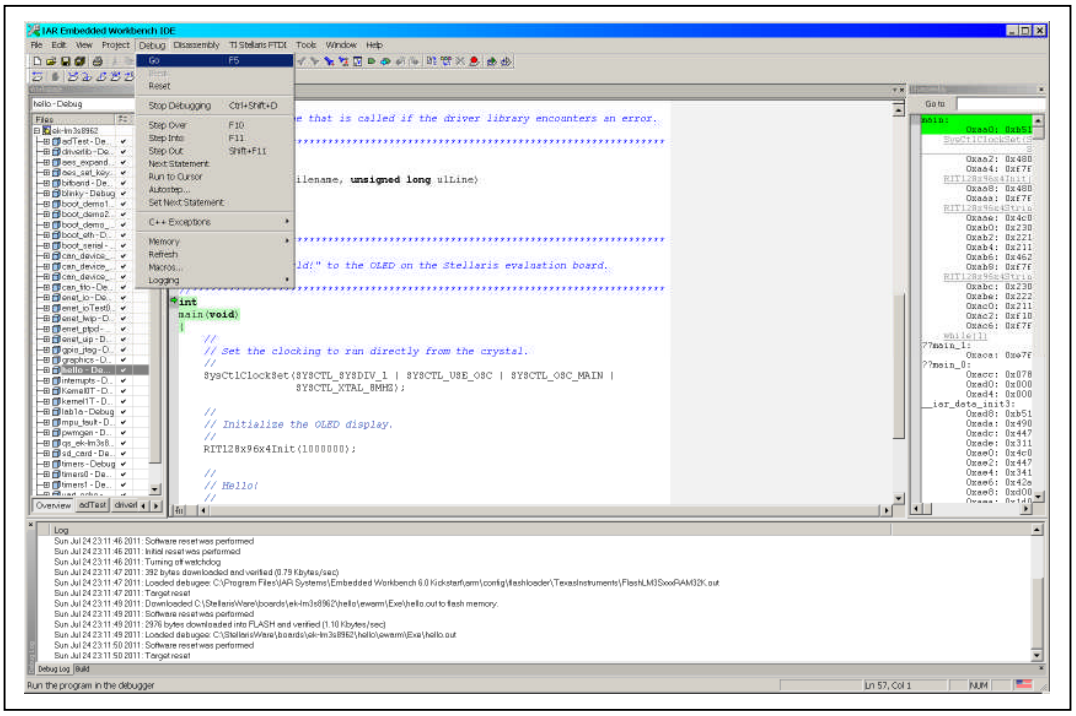
As our first step, we will first work with the IAR Embedded Workbench tools to build and run an existing simple program on the EKI-LM3S8962 chip.

The source code for several of the programs we will be using is in the same directory as the project assignment.

Working with the Tools

Building and Running a Program

1. Starting with step 3 of the Stellaris Quickstart – IAR Tools document,
   - ✓ Compile – select *Make* or *Rebuild All* from the *Project tab*
   - ✓ Download to the target board – select *Download and Debug* from the *Project tab*.
2. When the code has been downloaded to the target board, verify that the debugging halts with the cursor sitting on the start of the main routine.

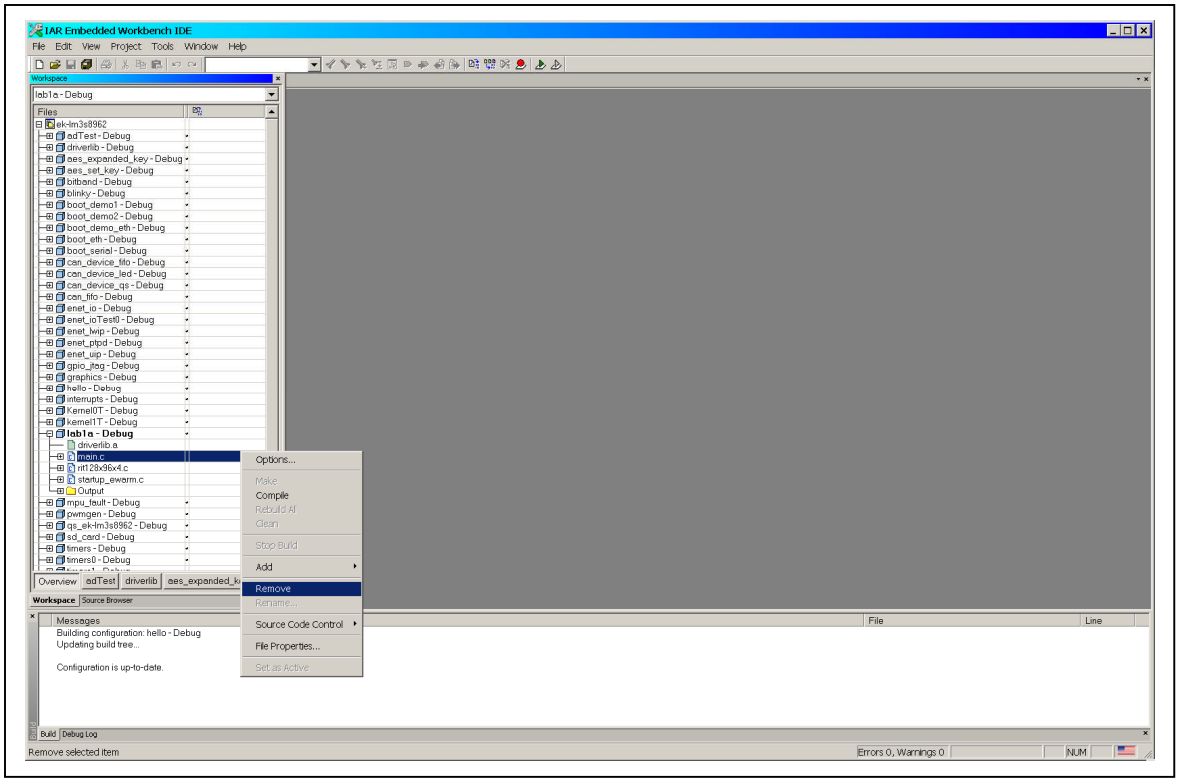


3. To run the program, select function key F5 or *GO* from the *Debug* tab. Verify that you have displayed *Hello* on the OLED.

Building Your Own Program

1. Starting with the section: *Creating a New Project* in of the Stellaris Quickstart – IAR Tools document, complete all of the specified steps to create a new project.

2. When you have completed creating your new project, it should appear as the currently active program in the *Workspace* pane (left hand window) of the IAR IDE.
3. Expand the project to show its contents. You should have:
   - ✓ driverlib.a
   - ✓ main.c
   - ✓ rit1128x96x4.c
   - ✓ startup_ewarm.c
4. Select main.c, open the right mouse menu, and select *Remove* to delete the file from the project.



5. On your PC, find the directory and project that you've just created. Delete the file main.c from that project.
6. Find the file project1a-2014.c in the Lab1 folder and copy it to your project directory on your PC and rename it main.c.
4. In the IDE, right click on the project and select Add > Add Files…
5. Add the new main.c to your project.
6. Annotate each line of the program to identify its purpose in the program and what it does (these really are different).
7. Compile your program and download it to the target board. There should be no errors – if there are, then bad copying.
8. Select *Download and Debug* from the *Project tab*

9. To run the program, select function key F5 or *GO* from the *Debug* tab.

The program should count in decimal according to the following pattern. Note that the following is actually 20 iterations through the program and the program's output will appear on only two lines on the OLED….at time t0, the value 9 is displayed…at time t1, the values 9 8 are displayed and so forth.

	The value of i is:
t0:	9
t1:	9 8
t2:	9 8 7
t3:	9 8 7 6
t4:	9 8 7 6 5
t5:	9 8 7 6 5 4
t6:	9 8 7 6 5 4 3
t7:	9 8 7 6 5 4 3 2
t8:	9 8 7 6 5 4 3 2 1
t9:	9 8 7 6 5 4 3 2 1 0
t10:	8 7 6 5 4 3 2 1 0
t11:	7 6 5 4 3 2 1 0
t12	6 5 4 3 2 1 0
t13:	5 4 3 2 1 0
t14:	4 3 2 1 0
t15:	3 2 1 0
t16:	2 1 0
t17:	1 0
t18:	0
t19:	
t20	9

10. Modify the program to parameterize the two *delay()* function calls in the two *for* loops so that they will support different user specified delays rather than the single hard coded value as they are now. Where will those values have to be specified?
11. Modify the program so that each of the two respective for loops is replaced by the following functions.

```
void f1Data(unsigned long delay1);
void f2Clear(unsigned long delay2);
```

12. Modify the program so that parameter, delay, is passed into the two functions in part 4 above by reference rather than by value.
13. Modify the program so that the two functions in part 12 above are replaced by a single function. The function should be able to be called with the character to be displayed and the value of the delay.
14. Modify the program so that the function you wrote in part 13 is in a separate file. Your program will now be composed of two files.

Working With the Debugger

Learning to use a debugger is an invaluable skill to develop if one is planning to do any serious software development work. Wahoo, we actually have a debugger this quarter. The need to develop strong debugging skills is independent of whether one develops embedded or desktop applications.

Again from Wikipedia…

> The terms "bug" and "debugging" are both popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However the term "bug" in the meaning of technical error dates back at least to 1878 and Thomas Edison. Further "debugging" seems to have been used as a term in aeronautics before entering the world of computers.

The IAR development environment provides some rather effective tools that help you to debug your code running on the LM3S8962. You have already worked with some useful features/abilities:

- setting and clearing breakpoints
- running to cursor
- single-stepping through or over code
- inspecting the contents of local/global variables
- inspecting and change the values of variables during runtime
- looking at how your high-level C code is translated to assembly

For each of the above items, there are many ways to perform the desired action. A brief summary on how to do each one is as follows:

- ***Breakpoints*** – To set a breakpoint, move the cursor to the left hand window border next to the desired line of code and left-click your mouse button. You can unset a breakpoint by simply repeating the process. Alternately, you can place your cursor on the desired line of code, then select '*run to cursor*' from either the main *Debug* or the right mouse menus. Personally, I prefer to '*run to cursor*'.

  Bear in mind that a breakpoint is *not* the place in your debugging process where you throw up your hands and go for a beer.

- ***Single-stepping*** – When in the *debug* mode, to single-step ***over*** each instruction in the code, press the F10 key. To step ***into*** a function (rather than over), press the F11 key.

Note that the code must be halted in the debug mode; it cannot be single-stepped while it is running.

Like breakpoints, single-stepping is not the process of slowly sneaking out when you've reached a breakpoint.

- ***Local/Global Variables*** – To inspect local/global variables, select either ***Add Watch*** or ***Quick Watch*** from the menu choices that appear. The elected variables and their values will appear in a window on the right hand side of the IDE. Explore a bit to see what the differences are.

Let's now put the debugger to work.

1. Build a project containing the code from the file project1b-2014.c in the Lab1 folder and rename it main.c. The program will compile and execute apparently correctly, however, there is a problem with it.

   Using your debugger, identify what the problem is and indicate how you found it with the debugger. Correct the problem and using your debugger, prove that you have, indeed, fixed the problem.

2. Build a project containing the code from the file project1c-2014.c in the Lab1 folder and rename it main.c. The program will compile, however, it has a problem during execution.

   Using your debugger, identify what the problem is and indicate how you found it with the debugger. Correct the problem and using your debugger, prove that you have, indeed, fixed the problem.

Building Your Own Applications

Now that you have successfully run an existing application on the target platform, this next exercise we will make a series of changes to the program to gain practice in some of the techniques that we will have to use in our more complex applications.

Application 1

Using what you have learned from the example programs in the previous exercises, write a program that will display the letters: A B C D on the OLED and flash them together at approximately a one-second rate.

Application 2

Modify the program in Application 1 to print then erase the letters A then B then C then D at approximately a one-second rate.

Application 3

Modify the program in Application 1 to flash the letters A and C at a one-second rate and the letters B and D at a two-second rate.

Implement the delay as a function with the following prototype:

```
void delay(int aDelay);
```

Deliverables

A project report containing

1. The annotated source code for all applications for the Cortex-M3 Processor.
2. Representative screen shots showing the results of executing the applications on the PC screen. Representative screen shots means that you need to visually document that you got the programs to compile and run on your system and that your design was able to meet the project requirements and specifications.