EGR424 S'10

Laboratory #1

Introduction

The purpose of this laboratory is to:

- establish a command-line development environment for the TI LM3S6965 processor and EKC-LM3S6965 evaluation kit.

  The CD that came with your evaluation kit has most of the necessary components, but it's not going to be a single one-click installation as is common for consumer-level hardware.

- download a simple "Hello World" application to the evaluation kit
- experiment with the different components of the toolchain

Briefly, the steps required for establishing the development environment are:

1. Install a command-line infrastructure
2. Install the CodeSourcery G++ Lite toolchain for the LM3S6965 processor
3. Install the TI StellarisWare firmware development package
4. Install device drivers for the evaluation kit
5. Install FLASH programming utility

Part I – Command-Line Infrastructure

Microsoft Windows provides graphical abstractions over underlying system services, such as windows, folders, and icons. Graphical user interfaces (GUI's) build upon these to provide abstractions over underlying software components. With abstraction, however, comes loss of control. As one of the themes of this course is "maximum control", we are not going to rely on these graphical abstractions but instead work at the command line, directly controlling the programs that run and the files they operate on.

While Windows does have a rudimentary command-line interface for controlling the computer (the "`cmd.exe`" program) it is not very powerful, it is not standardized across multiple operating systems, and it is missing several important utilities that greatly aid in software development.

We will be installing the Cygwin command-line infrastructure. This tool provides a "Unix-like" environment that is very similar to what users of Linux and Mac OS X (among other systems) already have available.

1. From `http://cygwin.com`, follow the link to "Install or update now!". This will simply download and execute a program named "`setup.exe`", which downloads the rest of the installation from the Internet. The direct link to this program is:

   ```
   http://cygwin.com/setup.exe
   ```

2. In the "Choose Installation Type" dialog box, select "Install from Internet".
3. In the "Choose Installation Directory" dialog box, proceed with the default "`C:\CYGWIN`" directory, and allow the installation to be available for all users.
4. In the "Select Local Package Directory" dialog box, proceed with the default "`C:\CYGWININSTALL`" directory.
5. Proceed to the "Choose Download Site(s)" dialog box. The Cygwin software components are replicated (*mirrored*) at several sites; choosing one close by is a good idea. The Oregon State University Open Source Lab is one of my favorites, so selecting `http://cygwin.osuosl.org` is recommended. Select this URL then click Next.

6. The setup program proceeds to download a list of all available software packages. You don't need to install all of them now – this would take a long time. You can, however, re-run this setup program at any time to update your Cygwin installation or install additional software packages.

7. The "Select Packages" dialog appears. The packages are broken down into broad categories, and default selections are made for each category. For the most part, the default selections are appropriate, but some changes have to be made:

   (a) Click on the word "`Default`" next to the "`Archive`" tree. This word should change to "`Install`". All of the packages in this category will now be installed.

   (b) Open the "`Devel`" tree. Click on the word "`Skip`" next to the "`gcc4-g++`" package (you'll have to scroll down). This should change to a version number indicating that this package will be installed. Note that this step also automatically selects other packages in this category, as they are dependencies for the "`gcc4-g++`" package.

   (c) In the same category, click on the word "`Skip`" next to the "`make`" package so that it will be installed.

8. The remaining categories have default selections that are appropriate for a first installation of Cygwin. Click on the Next button. Most likely, a "Resolve Dependencies" dialog will appear indicating that there are unmet dependencies. Make sure the checkbox at the bottom labelled "Install these packages to meet dependencies" is checked, then press Next.

9. Assuming a reasonably-fast Internet connection, the process should take about 15 minutes and consume about 700 MB of disk space. While waiting, you can begin on Part II of the laboratory. Eventually, you will see the "Installation Status and Create Icons" dialog. Clicking at least one of these options is suggested to easily start the Cygwin command line.

10. Try it out...double-click on the Cygwin icon on your desktop (or from the Start menu...whatever you selected in the previous step). You should soon be looking at a command-line window (often called a *shell*, or a *console*, or a *terminal* window – the latter is probably most correct). What to do now? Try some commands:

    (a) `ls`
    List the contents of the current directory (or what Windows likes to call a *folder*)

    (b) `cd /cygdrive/c ; ls`
    Change the current directory. The "`/cygdrive/c`" directory is the name Cygwin gives to the "`C:\`" root directory. The semicolon can be used to separate multiple commands. In this case, we change directories then list their contents with a single input line.

    (c) `cd ‘‘Program Files’’`
    Enter the `Program Files` directory...note that you need quotes around file and directory names that have spaces in them.

    (d) `cd ..  ; mkdir temp ; cd temp ; pwd`
    Go back up one directory (back to `/cygdrive/c`), create a new directory called "`temp`", enter that directory, then execute the "`pwd`" command which prints the current working directory just to remind us where we are.
    Let's stay in this directory for this lab...remember that it's equivalent to `C:\TEMP` in case you want to look at it from DOS or Windows.

If you are not already familiar with the Unix-style command line then try out some of the commands in the Shell Command Quick Reference documents available here:

`http://claymore.egr.gvsu.edu/~steriana/424/ShellQuickRef.pdf`

`http://claymore.egr.gvsu.edu/~steriana/424/ShellQuickRef2.pdf`

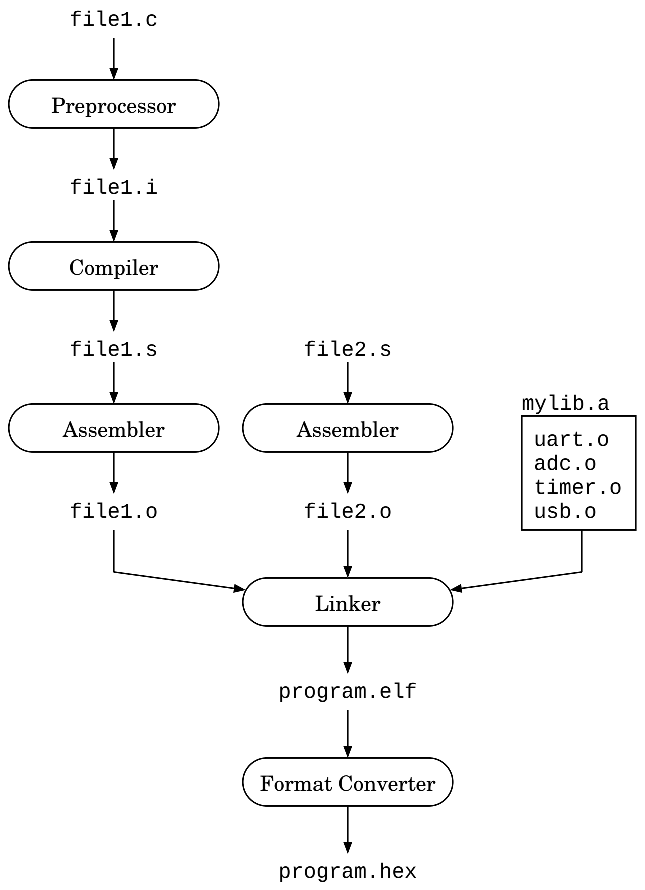

Figure 1: A minimal toolchain comprises a preprocessor, compiler, assembler, linker, librarian, and format converter which all work together to transform C (or other language) source files into a single file that can be programmed into an embedded system target. A program may comprise multiple source files, assembly language files, and pre-compiled files stored in libraries.

Part II – CodeSourcery G++ Lite Toolchain

Overview

The compilation of an embedded C program from source down to file-to-upload-to-FLASH has a lot of hidden steps and a lot of magic. An overview of this process is shown in Figure 1. One of the benefits of the open-source GNU development environment is the ability to look at and control each individual step of the process, thus minimizing the amount of "magic" often present in commercial development environments (which have the same steps, just not always available to study).

The collection of programs (tools) and libraries that allow all this magic is known as a "toolchain". Commercial companies (e.g., CodeSourcery, Keil, Code Red, IAR) compete based upon the quality and breadth of their toolchains, as well as other value-added differentiators. Differentiators may include:

- Speed of compilation for large projects
- Quality of generated code
- Adherence to the latest language standards (e.g., C99, C++0X)
- Breadth and depth of standard and extension libraries
- Support for specific hardware boards
- Graphical "wizards" for various parts of the development process
- Graphical user interface for development

- Hardware support for in-system debugging
- Support for simulation
- Technical support, community forums, ecosystem, multi-platform availability
- Of course, cost

CodeSourcery builds upon the open-source GNU toolchain to include GUI's, hardware support, etc. and sells various editions of their "Sourcery G++ Development System" with varying levels of features and support. Other companies (e.g., Keil) do not use the GNU toolchain as the foundation but instead have their own proprietary compilers and tools. For two reasons we will NOT be using the graphical CodeSourcery tools that come with the EKC-LM3S6965 evaluation kit:

1. They are only usable for 30 days, after which they cease to function until one of their commercial offerings is purchased
2. As one of the themes of the course is "no magic", you will want to see what is going on at each stage of the compilation process, not have it abstracted away by a graphical user interface.

Installation

1. From the EGR424 Wiki site:

   ```
   http://claymore.egr.gvsu.edu/egr326/LM3S6965
   ```

   go to the "Toolchain: Windows and Linux" section and click on the "IA32 Windows Installer" link. This is a 66.5MB download.

   CodeSourcery releases new versions of their tools every few months. The link on the Wiki site points to the "2010 Q1" release which is the latest release as of this writing and is based on GCC 4.4.1.

   Note the links directly underneath that point to several documentation files. Make a note to download these at some point in the future.

2. Follow the installation wizard accepting the defaults at each step ("Just keep clicking....just keep clicking...."[1])

3. To test it out, close your current Cygwin command line window and open a new command line window. This serves to reinitialize the path (the list of directories searched when you type in a program name) which is necessary to find the new toolchain programs you just installed.

   Type the following at the command line:

   ```
   echo $PATH
   ```

   Among the large amount of text shown, you should find the string:

   ```
   /cygdrive/c/Program files/CodeSourcery/Sourcery G++ Lite/bin
   ```

   This is the directory containing the toolchain programs.

4. Let's check the compiler's version. Type:

   ```
   arm-none-eabi-gcc -v
   ```

   The compiler program is "`arm-none-eabi-gcc`" and the "`-v`" flag tells it to print out configuration and version information. The very last line of the output shows that the compiler is GCC version 4.4.1.

[1] This is a reference to "Finding Nemo" in case you are curious

The Preprocessor

The first stage in the compilation process is preprocessing. The standard C preprocessor program is known as CPP (or `arm-none-eabi-cpp` for cross-compilation). It is a relatively-simple program that can include other files in the input (using `#include`), perform string replacements (using `#define`), or slightly more complicated parameter-based macros (also using `#define`). It can also perform conditional compilation using `#if`/`#else`/`#endif`.

For more information on the CPP preprocessor:

- type '`man cpp`'
- type '`info cpp`'
- browse to `http://gcc.gnu.org/onlinedocs/gcc-4.4.1/cpp`

1. Type in the following program to a file called `lab1.c`. Use a text editor such as GVIM (Notepad will do in a pinch but is not recommended).

```
#include <stdlib.h>

#define TEST_STRING "12345"
#define NUMBER_BASE 10

int gResult;

void main(void)
{
  gResult = strtol(TEST_STRING, 0, NUMBER_BASE);
}
```

2. The preprocessor can be invoked as a separate program (`arm-none-eabi-cpp`) or through the GCC driver program. The latter is recommended.

   Type the following:

```
arm-none-eabi-gcc -E -v -o lab1.i lab1.c
```

   The flags are as follows:

   - `-E` tells the GCC program to only preprocess the C source file, not to try to compile or assemble it
   - `-v` tells the GCC program to be "verbose" and display all of the intermediate programs it runs
   - `-o` sets the output file name. Traditionally, preprocessed-only source files are given the extension `.i` though this is not a requirement.

   Take a moment to study the verbose listing of what the GCC tool says when you run the above command. This information can be useful when debugging unexpected toolchain behavior.

3. Note that `arm-none-eabi-gcc` actually invokes another program known as CC1, hidden deep within the CodeSourcery directory. This program is the actual compiler that does all the "heavy lifting". The GCC program is actually just a driver program that co-ordinates all of the subprograms.

   Even though it is true that the name of the preprocessor program is CPP, the CC1 program actually has a copy of CPP built-in to it, hence it does not need to invoke it externally. The `arm-none-eabi-cpp` program is provided just as a convenience for people that want to use it for other purposes.

4. Take a few moments to view the `lab1.i` file now using a text editor such as GVIM. Note how your 10-line source file has ballooned into around 500 lines of code. Why? Because of the line at the top that says '`#include <stdlib.h>`'. This file, and all of the files that it includes, and all the files included in turn, are brought in verbatim. That's what the `#include` directive does, after all.

   Note, too, that the `lab1.i` file contains both actual C code and additional directives (beginning with `#`) that tell the next tool in the chain (the actual C compiler) where each line of code comes from (source file and line number). This information is useful for generating debuggable code.

5. Scroll down to the bottom of the `lab1.i` file. Note that:

   - your code appears here at last
   - there is no mention of `stdlib.h`...it has been pasted into the file (along with all the files it includes)
   - the `TEST_STRING` and `NUMBER_BASE` macros in the C source file have been expanded

   We included the `stdlib.h` file so that the `strtol()` C library function would be properly declared. Look through your `lab1.i` file until you can find the declaration of this function prototype. Scroll up from that until you find the #-directive that tells you which included file contained that declaration. For example, it may read something like:

   ```
   # 66 "c:\\program files\\codesourcery\\sourcery g++
   lite\\bin\\../lib/gcc/arm-none-eabi/4.4.1/../../../../arm-none-eabi/include/stdlib.h'' 3
   ```

   Open that file in an editor and verify that it contains the definition of `strtol()`. Now you know where the actual `stdlib.h` file resides on your system.

The Compiler

After preprocessing, the actual C compiler is invoked to convert C code to assembly code. For more information on the GCC compiler:

- type '`man gcc`'
- type '`info gcc`'
- browse to `http://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc`

1. Type the following (press up-arrow in your command shell to recall/edit previous commands and save typing):

   ```
   arm-none-eabi-gcc -Wall -O3 -march=armv7-m -mcpu=cortex-m3 -mthumb -mfix-cortex-m3-ldrd -S
   -v -o lab1.s lab1.i
   ```

   The `lab1.i` file that we got from the preprocessing step is now the input to the compiler, and `lab1.s` is the output (once again the output filename is controlled with the `-o` flag). Here is a description of the other flags:

   - `-Wall` indicates that all warnings should be displayed. This is a good idea in general as warnings can often indicate coding errors that are too subtle to be considered compiler errors. Some warnings (like `main()` not returning an `int`) can, of course, be ignored.
   - `-O3` indicates maximum speed optimization
   - `-march=armv7-m` tells the compiler to generate assembly code for the ARMv7-M architecture.
   - `-mcpu=cortex-m3` tells the compiler to specifically generate code for the Cortex-M3 processor family.
   - `-mthumb` tells the compiler to generate Thumb code. Since we also specified the ARMv7-M architecture and Cortex-M3 processor, the GCC compiler knows that it can generate Thumb-2 assembly language.
   - `-mfix-cortex-m3-ldrd` tells the compiler to work around a bug in the Cortex-M3 design (see issue #602117 in the ARM Errata document available on the course Wiki) by ensuring a particular sequence of assembly language instructions does not occur but is instead replaced with equivalent code that does not trigger the bug
   - `-S` tells the compiler to only generate assembly code, otherwise it will continue to the subsequent steps (which will follow shortly).

   Since the GCC compiler can handle a wide variety of processors and architectures the "`-m`" flags are needed to tell it the specific ones we are targeting. Also, we are now using the `-S` flag to indicate to the GCC driver program that we want to stop the compilation process at the assembly language level. The input to this compilation step is the preprocessor output from the previous section. The output is the file `lab1.s`. Note that the `.s` extension is fairly commonly used to indicate assembly files, and you should follow this convention.

2. Take a moment to look through the various messages displayed during the compilation process. Once again, the CC1 program was invoked to compile the source file but since it had a `.i` extension, the CC1 program treated it as a pre-processed file (can you see the flags given to CC1 that indicate this?)

   NOTE: Most programmers do not write code for embedded systems but instead write code that runs on the same computer they are working at. They would then be working with a "native compiler" (`gcc` for example). A compiler that runs on one computer but generates code intended to run on a different computer (or embedded system) is known as a "cross compiler" (`arm-none-eabi-gcc` for example). By convention, the name of the compiler tool itself gives you a clue as to what the target system is (in our case, an ARM system, no specific family, ARM EABI binary interface compatibility).

3. Have a look at the `lab1.s` assembly file generated. This output represents a significant amount of intellectual work expended by the compiler. The remaining parts of the toolchain are all fairly mechanical translators and processors, but the compiler proper is a substantial piece of software that is not easily written.

   The actual contents of the `lab1.s` file will make more sense as you become more familiar with the ARM Thumb-2 instruction set, but for the moment remember that the purpose of a C compiler is to translate C code to the assembly language specific to the target processor.

The Assembler

Now that we have an assembly file, we invoke the GNU assembler to translate assembly language mnemonics to actual 16-bit and 32-bit binary numbers that represent instructions. For more information on the GCC assembler:

- type '`man as`'
- type '`info as`'
- browse to `http://sourceware.org/binutils/docs-2.19/as/index.html`

1. Type the following (press up-arrow in your command shell to recall/edit previous commands and save typing):

   ```
   arm-none-eabi-gcc -Wall -O3 -march=armv7-m -mcpu=cortex-m3 -mthumb -mfix-cortex-m3-ldrd -c
                                   -v -o lab1.o lab1.s
   ```

   Once again, notice that we are not invoking the assembler directly but doing so through the top-level driver program `arm-none-eabi-gcc`. This is the preferred method of invoking all of the component tools as the driver program takes care of automatically adding other command-line flags that might be implied by the ones we specify.

   NOTE: Since you're probably getting tired of typing this long command line every time, take advantage of the command-line shell's aliasing features:

   ```
   alias CC='arm-none-eabi-gcc -Wall -O3 -march=armv7-m -mcpu=cortex-m3 -mthumb -mfix-cortex-m3-ldrd'
   CC -c -v -o lab1.o lab1.s
   ```

   You can also put the '`alias`' line in the `.bashrc` file in your home directory (just type '`cd`' with no parameters to get to your home directory) so it is all ready for you to use every time you open up a new window.

2. The `-c` flag is used to tell the GCC driver program to stop when it generates the object code (i.e., the 16-bit/32-bit numbers that represent instructions). Otherwise, it would continue on to try to generate a complete program. As before, the `-v` flag shows a verbose listing of everything that's going on. Can you see where the "`as.exe`" program was invoked to do the actual assembly?

3. Let's take a look at the `lab1.o` object file generated by the assembler:

   ```
   less lab1.o
   ```

   Ewwww....it's a binary file (in *ELF format*[2]). No matter. The GNU tools provide several convenient programs for working with object files.

[2]The Executable and Linking Format (ELF) file contains object code, debugging info, is processor and architecture independent, and has a lot of support across many systems. Its Windows counterpart is the EXE file. Interestingly, one possible format for storing the debugging info in an ELF file is known as DWARF.

Object Files

1. Type the following:

   ```
   arm-none-eabi-objdump -d lab1.o
   ```

   The `-d` switch displays the contents of all instructions (i.e., executable sections) nicely formatted as an assembly listing.

2. Note that when you compile your programs with the `-g` flag (enable debugging), then using `-S` instead of `-d` to dump the assembly code intermixes C source code with assembly....very nice! Try it now (we have to start by compiling the `.c` file for this to work):

   ```
   CC -c -v -o lab1.o -Wa,-a=lab1.lst -g lab1.c
   arm-none-eabi-objdump -S lab1.o
   ```

   Did you notice that the output of `arm-elf-objdump` actually put the two `#define` constants back in, even though they were taken out by the preprocessor? Nifty.

3. Also notice the new "`-Wa,-a=lab1.lst`" flag we added. The "`-Wa`" flag is a method of passing a command-line flag directly to the assembler. In this case, we are telling the GCC driver to call the AS assembler program with the "`-a=lab1.lst`" flag, which tells the assembler to generate a fancy assembler listing.

   Go ahead and look at the `lab1.lst` file now: it's got line numbers, C code, instruction encodings, etc. and is a useful thing to study when first learning assembly language.

4. Want to see some raw data? Try using using the `arm-none-eabi-objdump` program with the `-s` flag instead of `-d`. This is how your program will look in the FLASH memory of the ARM chip...just a bunch of 16-bit and 32-bit numbers.

   ```
   arm-none-eabi-objdump -s --section=.text lab1.o
   ```

   Why the "`--section=.text`" option? To only see the actual program code (known as the *text*). Try the `arm-none-eabi-objdump` program without this option to see the 32-bit numbers representing all the data in the ELF file.

5. Let's take a look at this object file's *relocation records* (each record indicates a piece of information that is not yet known but must be resolved before the final program is created):

   ```
   arm-none-eabi-objdump -r --section=.text lab1.o
   ```

   There are five relocation records. The third one is interesting because it is for `strtol` at offset 0x0000000E (the number is not important...yours may be different). This relocation record says that the instruction at this address has to be "fixed up" by the linker using an R_ARM_THM_CALL format; essentially this means that in order to create your final program, somebody somewhere is going to have to tell GCC where this `strtol()` function is in memory. Your `lab1.c` program doesn't define the body of this `strtol()` function...it just calls it. The relocation record for `strtol()` tells subsequent tools that this file you just compiled needs to be modified ("fixed up") once the final location of `strtol()` in FLASH is known. This is a job for the *linker* (see below).

Libraries

Referring back to Figure 1, you see that the final step in creating a program is the linker. The linker accepts pre-compiled object files (`.o` files) and stitches them together, placing functions and variables in well-defined memory locations, then fixing up all relocation records to finalize assembly language instructions that depended upon these addresses in memory.

The object files can come from two places:

- Object files created by compiling our .C source files
- Object files in a *library*

A library is nothing more than a collection of previously-compiled object files, all packaged up together into a single *archive file* (library file) which traditionally has a '`.a`' extension.

Remember that each object file generates several relocation records telling the linker where to fix up undefined references to external functions and global variables. Each of these relocation records is a trigger that tells the linker to search libraries, looking for an object file that defines the label sought by this relocation record. When the linker finds an appropriate object file, it pulls it in to the program, but this may generate additional relocation record requests, and the linker has to recursively repeat this process until all relocations are fixed up.

For example, our program generates a call to `strtol()`. The linker would:

- Search the standard C library (called `libc.a`) and find an object file named `strtol.o` which defines the label '`strtol`'
- The `strtol.o` object file is added to the final program, at some memory location. But this object file calls another function `_convert()` which generates another relocation record. The linker now scans the C library again looking for a module that defines the `_convert()` label, and so on.

The GNU library management program is named AR (for ARchiver). We will work with it more in the next laboratory.

Linking

At last it's time to create our final program. We must take our object file and link it together with other object files (there are no other object files in our simple one-file case) and with the C library. More information on the GNU linker is available by:

- typing '`man ld`'
- typing '`info ld`'
- browsing to `http://sourceware.org/binutils/docs-2.20/ld/index.html`

1. Type the following:

   ```
   CC -v -nostdlib -o lab1.elf lab1.o
   ```

   Note that the GCC driver program invoked a program named COLLECT2, which is the linker program. This program is normally called just LD but COLLECT2 is slightly more complicated in order to support C++.

   Unfortunately, our linking process did not work. The last line that says that "`ld returned 1 exit status`" is an error message. Under Unix-type systems, programs return 0 to indicate success and anything else to indicate failure.

2. The problem is that we passed the `-nostdlib` switch to the GCC program. Fooled you!!! This switch tells GCC to **not** try to link in the standard C library. Look carefully at the output of the command that you typed and you will note the error messages:

   - "warning: cannot find entry symbol _start; defaulting to 00008000"
   - "undefined reference to 'strtol'"

   Why is the linker looking for the symbol `_start`? Doesn't our program start at `main()`? No, it doesn't. Lots of code is executed before `main()` is ever reached. The very first instruction that the device is to execute after powering on is to be placed at the label called '`_start`'. This label is actually pointing to code that's in the standard C library.

   Finally, it should be no surprise that there is an undefined reference to `strtol()`. This is a standard C library function and we expressly told the C compiler (using the `-nostdlib` option) to not look in this library.

The Standard Library and Startup Files

1. At last, let's let the linker use the standard C library by removing the `-nostdlib` flag:

   ```
   CC -v -o lab1.elf lab1.o
   ```

   If you can pick your way through the multiple lines of output, you will notice that the linker is stitching together the following object files (the order is important), even though you didn't ask for most of them:

   - `crti.o`
   - `crtbegin.o`
   - `lab1.o`
   - files from the GCC and C libraries (specified with the `-lgcc` and `-lc` flags)
   - `crtend.o`
   - `crtn.o`

   There's lots of magic here. We will peel back the layers of magic some other time. For the moment, take note of the `-L` and `-l` flags passed to COLLECT2. The `-L` flag tells the linker what directory to search for library files (which have a `.a` extension). The `-l` flag tells the linker which library file to search for resolving relocation records. Note that specifying `-lc` tells the linker to look for the file `libc.a` in any one of the directories specified with an `-L` flag. The linker automatically prepends '`lib`' and appends '`.a`' to whatever you specify with the `-l` flag.

   One of the utility libraries included with many GNU projects is called `libiberty.a` so that it is linked with the flag `-liberty`. I am not making this up.

   Notice that two libraries are being searched: `libc.a` (`-lc`) and `libgcc.a` (`-lgcc`). The `libc.a` library has the standard C library functions (like `strtol()`) while the `libgcc.a` library has "helper" functions that you should never need to call yourself. For example, `libgcc.a` contains the function `__aeabi_ddiv` which implements floating-point division as a subroutine. Try looking at the assembly source code for the C function:

   ```
   double f(double i, double j) { return i/j; }
   ```

   and you will see that an external `__aeabi_ddiv` function is used to perform the actual division. This function is pre-compiled and stored in the `libgcc.a` library.

2. Where are these libraries on the computer? Let's ask the compiler:

   ```
   CC -print-file-name=libc.a

   CC -print-libgcc-file-name
   ```

ELF Files

So now we have an ELF file...our final program. What can we do with it?

1. We can use `arm-none-eabi-objdump` to inspect ELF file contents, just like we did with `.o` files. Try this:

   ```
   arm-none-eabi-objdump -d lab1.elf | less
   ```

   Wow, lots of code! This is the assembly code for the whole program, most of which comprises functions from the standard C library. Scroll down a bit until you find your `main()` function. Notice that the '`bl strtol`' instruction has finally been fixed up by the linker to point to the actual numeric memory address 0x81A0 (yours may look a bit different):

   ```
   804e:      f000 f8a7      bl      81a0 <strtol>
   ```

2. Here's another way to look at the final addresses of the program's labels:

```
arm-none-eabi-objdump -t lab1.elf | less
```

See if you can find where in memory your `main()` function will be located. This might be easier:

```
arm-none-eabi-objdump -t lab1.elf | grep main
```

3. We can also use the NM program to quickly list the symbol table from our ELF file:

   ```
   arm-none-eabi-nm lab1.elf | grep main
   ```

   ```
   arm-none-eabi-nm lab1.elf | grep gResult
   ```

   This tells us that the `main()` function is located at address 0x00008040 in memory while the global variable `gResult` is at address 0x00010664 (probably...it may vary depending upon the version of the toolchain you are using).

4. We can use the SIZE program to see how big the program is:

   ```
   arm-none-eabi-size lab1.elf
   ```

   This gives us the size of each section (we haven't discussed sections yet). Roughly speaking, if you add up the numbers in the `text` and `data` columns you will see how much FLASH (i.e., program) memory is required. If you add up the numbers in the `data` and `bss` columns, you will see how much RAM is required. The `dec` and `hex` columns are decimal and hexadecimal representations of the sum of all three columns (`text`+`data`+`bss`) which isn't really that useful.

5. We can also strip unnecessary information from the ELF file:

   ```
   arm-none-eabi-strip lab1.elf
   ```

   Try looking for the `main` or `gResult` symbols using NM or OBJDUMP as you did above. They're gone...the STRIP program took out everything that isn't strictly necessary for running code, such as the symbol table and debugging information.

   The ELF file will get smaller when you strip it, but this isn't really a big deal given the size of ELF files we will deal with. On your native system, however, big programs can lead to a large amount of disk space usage if left unstripped. Try finding the CC1 program in your CodeSourcery directory and look at how big it is:

   ```
   find /cygdrive/c/Program\ Files/CodeSourcery -name cc1.exe -exec ls -al {} \;
   ```

Other Formats

The other thing we can do with ELF files is convert them to other formats, most likely for downloading to a hardware target system.

1. Try this:

   ```
   arm-none-eabi-objcopy -O ihex lab1.elf lab1.hex
   ```

   Open the `lab1.hex` file in GVIM. If you don't see a pretty colorized display, type:

   ```
   :set syntax=hex
   ```

   The `arm-none-eabi-objcopy` program converted the important parts of the ELF file (code and data only) into Intel HEX format, which is understood by many hardware programmers (this is the format understood by AVR Studio, AVRDUDE, etc.) An Intel HEX file is a text file that comprises line-oriented data records. Each line (data record) contains:

- a colon ':' as the first character
- two hex digits indicating how many data bytes are in this data record (e.g., '10' means 16 bytes)
- four hex digits indicating the address where the data in this record should be loaded into FLASH or RAM
- two digits indicating the record type. A record type of 0 indicates data, type 3 indicates the start address of the program, and type 1 indicates end-of-data.
- groups of 2 hex digits indicating data
- a final 2-digit checksum over the entire line

For more details on the Intel HEX file format, download the Intel specification from:

```
http://www.precma.it/download/intelhex.pdf
```

This format is a bit outdated because addresses in the file are only 16 bits, while the ARM supports 32-bit addresses. This is handled by "extended linear address" records in the HEX file that set a 16-bit prefix to subsequent 16-bit addresses, hence forming full 32-bit addresses.

2. In the worst case, we can just convert the ELF program to a binary file, exactly the way it would look in the ARM chip's memory, then just send the file as-is to some kind of bootloader on the chip.

```
arm-none-eabi-objcopy -O binary lab1.elf lab1.bin
```

Let's take a look at it:

```
od -t x4 lab1.bin | less
```

The 'od' program is a standard Unix utility (it stands for Octal Dump) that prints out the contents of binary files in various formats. The "-t x4" flag says to print out each 32-bit word in hexadecimal format.

Are We There Yet?

So we have a complete program...can we send it to the EKC-LM3S6965 evaluation kit and watch it run?

No. If you look carefully at the output of the linking phase:

```
CC -o lab1.elf lab1.o
```

you should see the warning:

```
warning:  cannot find entry symbol _start; defaulting to 00008010
```

The GCC toolchain knows about ARM Architectures and Processors; it does not know about Devices. It doesn't know where there is FLASH, where there is RAM, where all the hardware I/O addresses are for the UART's, the USB interface, etc. etc. nor does it know what hardware devices are available on-chip. The warning you see above is essentially GCC's way of telling you "I have no idea what physical addresses represent FLASH and RAM on your device so I will just arbitrarily start your code at 0x8000....hope that works out for you.....g'bye".

Unfortunately, that's not going to work for us. We will need to provide a *linker script* to tell the linker these specifics: where is FLASH? where is RAM? how big are they? etc.

In addition, we are going to want to access the on-chip hardware so that we can interact with the evaluation kit. Look at what the program does now: it sets the value of a global variable then exits. How can we blink some LED's? respond to pushbuttons? display something on the OLED display? The GCC toolchain has no responsibility there, so we turn to our next tool, the Texas Instruments StellarisWare Firmware Development Package.

Part III – StellarisWare

As described above, GCC stops at the generation of instructions....it doesn't know about the actual hardware you're working with. Texas Instruments helps you out by providing the StellarisWare firmware development package, which comprises C libraries, linker scripts, etc. to help you finish the task of writing code specifically for the LM3S6965 processor, and even more specifically for the EKC-LM3S6965 evaluation kit.

1. There are two ways to install StellarisWare:

   (a) Directly from the CD that came with your evaluation kit (follow the links for installation, or look in the `Tools/StellarisWare` directory of the CD).

   (b) On-line from Texas Instruments (see the Software Development section of the course Wiki). This will give you the most recent version of these tools, but since the tools are export-controlled and you must agree to their terms and conditions, no direct link to the tools can be posted on the Wiki. Fortunately, agreeing to the terms and conditions can all be done on-line and you should receive a direct link for downloading by e-mail from Texas Instruments within minutes of completing the process.

   Whichever way you decide to go, start installing! Maybe go with the CD route for expediency and make a note to download the more recent version from TI's web site at some point.

   Accept all the defaults to install to the `C:\StellarisWare` directory.

2. Let's poke around StellarisWare to see what they provide:

   (a) `pushd /cygdrive/c/StellarisWare`
   The '`pushd`' command is like '`cd`' in that it enters the given directory, but also saves the current directory on a stack so we can come back to `/cygdrive/c/temp` where our files are using '`popd`' (we'll do this shortly after we finish our visit of StellarisWare).

   (b) `ls boards/ek-lm3s6965/hello`
   This is a sample application that, when compiled, will print "Hello World" on the OLED display. We're going to be compiling a simplified version of this shortly. Note that there are several other sample applications in the `boards/ek-lm3s6965` directory.

   (c) `ls boards/ek-lm3s6965/drivers`
   This directory contains a driver for the OLED display specific to the EK-LM3S6965 board. It comes as a `.c` file and a `.h` file thus we will have to incorporate both of these into our own code if we want to use the OLED display.

   (d) `ls docs`
   Lots of documentation files here...the most current versions have links to them on the course Wiki.

   (e) `ls driverlib`
   This directory contains source code for drivers for the hardware on the LM3S6965 processor (what Texas Instruments calls the "Peripheral Driver Library"). These have already been compiled into `.o` object files and combined into a library which you can find at......
   `ls driverlib/gcc`
   There is the `libdriver.a` library that we will need to link with if we want to incorporate TI's StellarisWare library into our own code.

   (f) `ls inc`
   These header files need to be `#include`-ed into our `.c` files to provide prototypes of StellarisWare functions.

   (g) `ls third_party tools utils`
   Neat third-party drivers for things like AES encryption, FAT filesystem support, etc. Poke around if you're interested. We'll look at some of these later in the course.

   (h) `popd`
   This should return you back to the `/cygdrive/c/temp` directory (it undoes the "`pushd`" instruction we started with).

3. The summary of our investigation is that:

(a) The `.h` include files for the Peripheral Driver Library are in the following directory:
`C:/StellarisWare/inc`

(b) The `.a` pre-compiled library for the Peripheral Driver Library is in the following directory:
`C:/StellarisWare/driverlib/gcc`

(c) The name of the pre-compiled library is `libdriver.a`.

(d) The OLED driver is represented by a `.c` and a `.h` file in the `C:/StellarisWare/boards/ek-lm3s6965/drivers` directory.

The above mean that we are going to have to compile code like this:

```
CC -IC:/StellarisWare -IC:/StellarisWare/boards/ek-lm3s6965
    -LC:/StellarisWare/driverlib/gcc [.....]  -ldriver
```

- The `-I` flag for GCC tells it where to look for additional `.h` include files in addition to the ones it already knows about. We don't include the '`inc`' suffix on `C:/StellarisWare` nor the '`drivers`' suffix on `C:/StellarisWare/boards/ek-lm3s6965` because you will see that the code we are compiling uses statements like:

```
#include <inc/hw_adc.h>
#include <drivers/rit128x96x4.h>
```

- The `-L` flag for GCC tells it where to look for additional `.a` library files in addition to the ones it already knows about
- The `-ldriver` flag tells GCC to link with the `libdriver.a` library (note that `-l` is a lowercase L, not the number 1)
- The `[.....]` represents any other flags we specify, `.c` files, etc. The "`-l`" flags have to come at the end of the command. Among the C files we specify must be the OLED driver file (if we're going to use the OLED in our application).

4. Let's try it out with a real program. The "hello" program is a good place to start. If you look in that directory you will see a lot of files and a fairly complicated and generic build process. We're going to strip it down to something simpler and more understandable:

   (a) cp `/cygdrive/c/StellarisWare/boards/ek-lm3s6965/hello/hello.c .`
   cp `/cygdrive/c/StellarisWare/boards/ek-lm3s6965/hello/startup_gcc .`
   cp `/cygdrive/c/StellarisWare/boards/ek-lm3s6965/hello/hello.ld .`
   The '`cp`' command is used to copy files. The first two commands above copy the `hello.c` and `startup_gcc.c` from the "hello" sample application to our `/cygdrive/c/temp` working directory ('`.`' means "the current directory"). These two `.c` files represent "the application" while everything else comes from the StellarisWare library.
   The third command copies the `hello.ld` linker script which tells the linker where everything should go in memory. This linker script is specific to the LM3S6965 processor; we will study linker scripts in great detail later in the course.

   (b) `ls`
   Let's just make sure all the files we need are in the current directory.

   (c) Let's compile (put this all on one line – it's written on separate lines below for clarity):

```
CC
    -IC:/StellarisWare
    -IC:/StellarisWare/boards/ek-lm3s6965
    -LC:/StellarisWare/driverlib/gcc
    -o hello.elf
    hello.c startup_gcc.c
    C:/StellarisWare/boards/ek-lm3s6965/drivers/rit128x96x4.c
    -ldriver
    -T hello.ld
    --entry ResetISR
    -Wl,--gc-sections
```

Can you explain the meaning/purpose of each line above? You should be able to, except for the last 3 lines.

- The "`-T`" flag tells GCC to use the linker script `hello.ld`. Remember that this file tells GCC where it should place things in memory. Take a look at it now if you are curious. The MEMORY section of this file, for example, should be self-explanatory. It shows that this processor has FLASH memory at address 0x0 with length 0x40000 (256kB of FLASH) and SRAM at address 0x200000000 with length 0x10000 (64kB of RAM).
- The "`--entry ResetISR`" flag tells GCC that the program should start at the C function `ResetISR()` (this is in the `startup_gcc.c` file). This will be the very first function executed when the program starts, not `main()`, because it needs to set up the "C runtime environment" prior to invoking `main()`. More on this later in the course.
- The "`-Wl,--gc-sections`" flag passes the "`--gc-sections`" flag directly to the linker ("`-Wl`" is used to pass flags to the linker just like "`-Wa`" was used to pass flags to the assembler....and note that "`-Wl`" has a lowercase letter L, not the number 1). This option performs "garbage collection" on unused sections, which can greatly reduce the size of the program by removing code from the peripheral driver library that is not specifically used in the current application.

That's it...we're done. If your compilation was successful you should be able to see a "`hello.elf`" file in the current directory. This program has a good chance of working as expected on the evaluation kit.

Part IV – Device Drivers

1. Read the "Read Me First" document in the Documentation section of the installation CD. You can also find this file directly in the Documentation folder of the CD. Then, go ahead and follow the instructions there to plug in your evaluation kit and install the drivers.

   Note that these are standard FTDI drivers for their USB-to-serial converters, customized to recognize the VID/PID of the evaluation kit. Thus, the installation of these drivers should be familiar for anyone who has worked with devices such as the FT232RL.

Part V – FLASH Programming

We have our application ready to run, the last piece of the puzzle is actually installing it onto the FLASH memory of the LM3S6965 on the evaluation kit.

To do so we will install TI's LM Flash Programmer utility. This can be found on the installation CD in the Tools section, or directly as the `Tools\LMFlashProgrammer\LMFlashProgrammer.msi` file.

1. Go ahead and install the LM Flash Programmer tool now.

2. Start the LM Flash Programmer. Under the Configuration tab, in the Quick Set drop-down list select the LM3S6965 Ethernet Evaluation Board.

3. Click on the Program tab. The "Select .bin file" text box suggests we're going to need a file in the `.bin` (binary) format, when what we have is a `.elf` (ELF format) file. How can we convert from ELF format to a plain binary file? We did this earlier....do you remember?.....

   ```
   arm-none-eabi-objcopy -O binary hello.elf hello.bin
   ```

4. Back to the LM Flash Programmer, click on the Browse button and navigate to C:\TEMP\HELLO.BIN. Click the Program button and cross your fingers.

5. If there are no errors, press the Reset button on the evaluation kit (it's right next to the OLED) and the OLED display should show "Hello World!".

Postmortem

All of the tools, wizards, GUI's, etc. provided by CodeSourcery and TI are intended to make life for you as simple as possible and to hide the complexity. That's good in the long term, once you understand what's actually happening. But in this course we're going to embrace the complexity, do things the hard way, so that you do indeed understand what's actually happening. You can then use the abstract tools in confidence, knowing that you have mastered the tools, rather than been enslaved by them.

Mac OS X and Linux

Almost all components of the development environment described in this laboratory are available for Mac OS X and Linux (see the Wiki), except for the LM Flash Programmer utility which only runs under Windows. Techniques for sending programs to the FLASH memory of the evaluation kit under Mac OS X and Linux are under investigation, and hopefully then users of these operating systems will not need to use Windows at all for this course. Of course, the Cygwin environment is not needed on either Linux or Mac OS X.