Experiment 1:
Introduction to the Laboratory

This first experiment is one of the most important ones that you will be doing in ELEC2041 Microprocessors and Interfacing. It lays the foundation for everything that is to come, showing you some important principles and guiding you through some of the practical details of microcontroller-based systems.

Aims

This experiment aims to:

- introduce you to the DSLMU Microcontroller Board in the Digital Systems Laboratory,

- introduce the host environment in the Laboratory,

- show the fundamentals of editing, compiling and debugging a simple program on the DSLMU Microcontroller Board,

- illustrate "best practice" principles when writing source code, and

- introduce simple ARM assembly language instructions.

Preparation

It is *extremely* important that you prepare for each laboratory experiment, so that you can use your time (and your partner's time) most effectively. This particular experiment does not require much preparation; however, before coming into the Laboratory, you should:

- obtain one or two 3½˝ floppy disks on which to store your work,

- get your own copy of this Laboratory Manual, as well as the Companion CD-ROM,

- read through this experiment in detail, trying to understand what you will be doing, and

- quickly skim through *An Introduction to Komodo*, *An Introduction to the GNU Assembler* and at least the first four pages of the *DSLMU Microcontroller Board Hardware Reference Manual*. You can find these documents in the Appendices, as well as on your CD-ROM.

If you are keen (and you should be!), you could also:

- browse through this Laboratory Manual and the Companion CD-ROM,

- install the software on the CD-ROM onto your computer at home,

- type up or modify the necessary files in this experiment, to save time in class, and

- where possible, run through the experiment at home using the simulator.

Updates

Although this Laboratory Manual and the associated Companion CD-ROM should be all that you need, you should regularly check this course's Web site for any updates; you can find this site at *http://subjects.ee.unsw.edu.au/~elec2041/*.

Signing in

As mentioned in your lectures, you will be working with a partner during these experiments. When you arrive at the Laboratory for the first time, make sure you "sign in" with such a partner.

When you sign in with your partner, you will be given an account name to use for this course. Once you have this account name, move to any one of the 24 bench positions that are vacant.

The PC should be showing the Linux login screen; if not, consult one of the Laboratory staff. Log in using your new account name as the login name; you will be given the default password when you first sign in.

The first thing you should do is change the default password to something that both you and your partner can easily remember. To do this, click on the **Shell** icon in the top left-hand corner of the screen. A window will open, containing the command line prompt—in Unix terminology, the *shell*. Type in the following command:

```
passwd
```

Follow the prompts and, *voila!*, your password is now changed. It is best to test this, so log out by clicking the **K** icon (shown below), then selecting **Logout**:

By the way, this Laboratory Manual assumes for the most part that you have had some experience with Unix/Linux-like systems.

Equipment Handling Precautions

A few precautions are necessary in the Digital Systems Laboratory:

- The Laboratory is an environment where, unfortunately, it is quite easy for you to build up a static electric charge. **Electrostatic discharge can destroy electronic equipment**, including the circuits used in this Laboratory, without giving any sign of doing so! Please make *every* effort to discharge yourself before handling the DSLMU Microcontroller Boards, and observe any handling precautions as advised by the staff.

   Since you may be carrying a static charge without even realising it, **you should *always* discharge yourself**. You can do this painlessly by touching a grounded conductor using a coin, a key or another metallic object instead of your finger. You can use one of the oscillator input connectors as a suitable grounded conductor.

- Short circuits may damage certain devices. Please remove any metal watch straps, buckles, rings and so on *before* handling the boards. There are no dangerous voltages or currents that will harm you here, but this is a good habit to form for times that you may need to be more careful.

- Always turn the power *off* before connecting or disconnecting any I/O subsystems.

Examining the DSLMU Microcontroller Board

Now that you have your account name and password, log back in and take the time to examine the environment in which you will be working. This consists of three main pieces of equipment: the DSLMU Microcontroller Board,[1] the oscilloscope and the Host PC, the computer you have already been using. Figure 1 depicts the physical layout of the DSLMU Microcontroller Board:

---

[1] If you are wondering what on earth "DSLMU" stands for, it is an abbreviation of "Digital Systems Laboratory and Manchester University", the two entities responsible for the design of this board.
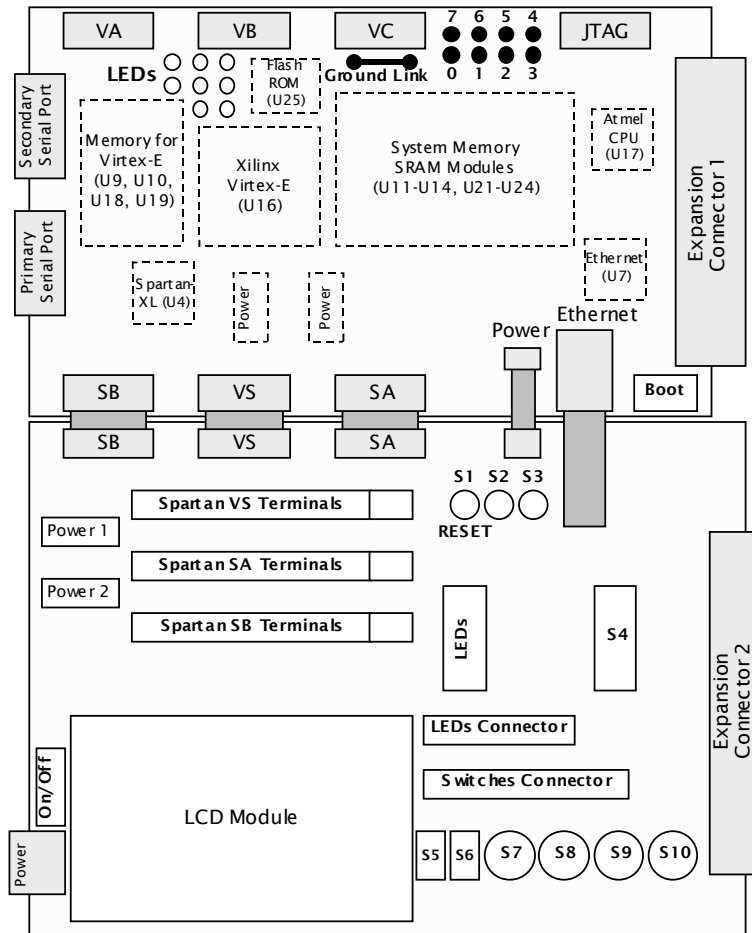
Figure 1: The Physical Layout of the DSLMU Microcontroller Board

As you can see from this figure, the DSLMU Microcontroller Board actually consists of two printed circuit boards connected to each other: the MU Board, which contains most of the electronics, and the Expansion Board, containing most of the peripherals and connectors. These boards were designed by staff at the University of Manchester, England, and the School of Electrical Engineering, University of New South Wales, Australia.
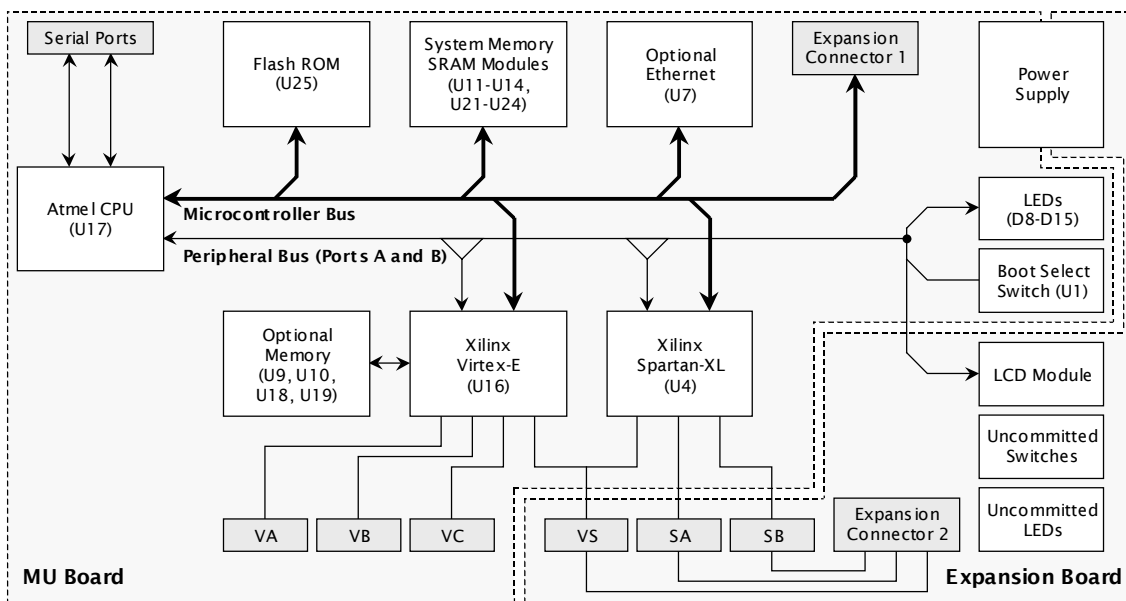
The DSLMU Microcontroller Board is based on an ARM 32-bit Reduced Instruction Set Computer (RISC) microcontroller and two Field Programmable Gate Arrays (FPGA). The particular microcontroller used on this board is the Atmel AT91R40008 device, one of dozens available from different manufacturers world-wide, all of which use the ARM core.[2] Each of these microcontrollers is electrically different and has its own set of capabilities; however, all are programmed using the same instruction set.

Why use ARM? As described elsewhere, this course attempt to teach you the *principles* of microprocessors and interfacing. However, it is also highly advantageous, from your own perspective, to use one of the most widely used modern computer architectures in the embedded industry—and ARM represents the largest player in the world's 32-bit embedded market. Indeed, that mobile phone you are carrying (and which should be switched off in the Laboratory!), that PDA you have in your pocket, that MP3 player in your back-pack—more likely than not, all of these are running on some variant of the ARM 32-bit RISC architecture.

---

[2] For the curious, ARM itself does not manufacture its own microcontrollers (it runs what is known in the industry as a "fab-less shop"). Instead, it sells the intellectual property (IP) rights to its design to the different manufacturers, who do the hard work of manufacturing the physical silicon. ARM maintains a list of these manufacturers; see ARM's Web site at *http://www.arm.com/* if you are interested in such details.

Apart from the ARM microcontroller, the DSLMU Microcontroller Board provides two Field Programmable Gate Arrays (FPGAs): the Xilinx Virtex-E and the Xilinx Spartan-XL. These FPGAs are designed to be reprogrammed to become any sort of peripheral that you might want (within limits, of course). For example, you could program the Spartan-XL FPGA to be a Real Time Clock (a device that keeps track of the actual date and time), or to be a latch-and-debounce circuit for input switches. The Xilinx Virtex-E is even powerful enough to be used as a custom coprocessor!

Figure 2 shows the system block diagram of the DSLMU Microcontroller Board. Study this diagram carefully, as it will help you understand how the board works:

Figure 2: System Block Diagram

You can find much more information on the DSLMU Microcontroller Board in the *Hardware Reference Manual*. You can find this Manual on your CD-ROM in the *board/doc* directory, or in one of the Appendices. Take the time to skim through that document!

Task: Take a closer look at the particular microcontroller used on the DSLMU Microcontroller Board: you will find as U17. Open to page 4 of the *Atmel AT91R40008 Datasheet Summary* and observe the block diagram for this processor (you can find this datasheet on your CD-ROM in the *board/doc* directory, or in an Appendix). Can you take a guess at what blocks were licensed from ARM and what blocks were added by Atmel itself? How simple or complicated do you think those blocks from ARM are compared to the ones added by Atmel?

Now, identify the following components on the two physical boards, observe them on the system block diagram (where applicable) and try to trace the connections between them and other major components:

- the main crystal oscillator, U3 which generates timing signals for the entire system,

- the static RAM components, U11–U14 and U21–U24, providing read/write memory, in which your programs will be stored and run,

- the Flash ROM component, U25, providing read-only storage, and which contains the programs necessary to communicate with the host PC,

- the Xilinx Spartan-XL FPGA, U4, containing up to 10,000 programmable systems gates, allowing you to design custom hardware for the board,

- the Xilinx Virtex-E FPGA, U16, containing up to 412,000 programmable system gates, allowing you to design a custom coprocessor for the board,

- the LCD module, which you can use to display messages,

- the 4-way DIP switch, which allows you to select the mode of operation for the board,

- the LEDs in the shape of "traffic lights", the bar-graph LEDs and various push-buttons and other switches, peripherals you will use in various experiments,

- the screw terminal connectors, to which you can attach your own peripherals,

- the On/Off power switch, which you will use whenever you need to attach peripherals to the board, and

- the Reset button (in red!), which you may need to use if your program crashes.

Examining the Oscilloscope

You should already know how to use an oscilloscope from previous courses. For this reason, all that you need to do for the moment is to take a quick look at the oscilloscope that is part of your Laboratory environment. Take the time to identify the two main input channels and the various controls associated with them. In particular, locate the *scale control*, which is marked in volts per (vertical) division, and the *time base control*, marked in seconds per (horizontal) division. You will be using these controls in later experiments.

Examining the Host Environment

Now turn your attention to the host PC to which the DSLMU Microcontroller Board is connected. This host PC is simply a standard personal computer running the Linux operating system and the KDE graphical environment; you will be using this environment to edit your source code files, compile these files into executables, download these to the DSLMU Microcontroller Board and debug and run them there.

You should see a fairly standard KDE desktop every time you log in. Take note of the desktop icons in the top left-hand and top right-hand corners:

Figure 3: Desktop Icons

The **Shell** icon, in the top left-hand corner, opens up a Unix shell (command line) window. You have already used this icon in this experiment; you are expected to know basic Unix commands from other programming courses.

The **Editor** icon starts the default text editor. In this laboratory you will be using the KDE Advanced Text Editor **kate**.[3] Take some time to familiarise yourself with this editor: click on the icon, watch the editor window come up, create a new file, place some random text into it, save the file (give it any name you like) and so on.

The two icons in the top right-hand corner are convenient short-cuts to the Companion CD-ROM and to the *ARM Architecture Reference Manual*. You are encouraged to use these resources during your Laboratory experiments—and you don't even need to bring your own CD-ROM in every time!

A Simple Program to Get Started

It's time to get your hands dirty, so to speak, with some programming. Since everyone likes to see something "real", preferably with flashing lights, your first program on the DSLMU Microcontroller Board will be to get the eight LEDs to flash on and off.

It is a good habit, and one that you should adopt, to create separate directories for each project. This translates to creating a separate directory for each experiment in this course.

Task: Given the above, open a shell window and type:

```
mkdir ~/exp1
cd ~/exp1
```

You should now be in the *exp1* subdirectory just beneath your own home directory (you can verify this with the Unix **pwd** command[4]).

Now, copy all of the files you will need for this experiment from the Companion CD-ROM. In the Laboratory, you can find a copy of the CD-ROM in *~elec2041/cdrom*. You can copy the files you need by typing:[5]

```
cp ~elec2041/cdrom/unsw/elec2041/labs-src/exp1/* .
```

Be careful to note the "**.**" at the end of the **cp** command! And don't close the shell window just yet as you will be using it extensively…

Program Specification

The first step in any program development is to come up with a *specification:* a description of what you want the program to achieve. Or, in real life, what your *customers* want the program to achieve. Remember the old English proverb, "an ounce of prevention is better than a pound of cure"! Trying to save time by skimping on the specification only wastes time later on during implementation.

In this case, the specification is simple: the program must flash the LEDs on and off. In other words, the program must first turn the LEDs on, then turn them off, repeating this process forever (or until stopped, whichever comes first).

If you consult the *DSLMU Microcontroller Board Hardware Reference Manual*, you will find that the eight LEDs on the MU Board can be accessed at address 0x10000000.[6] Writing a byte to this address will turn the LEDs on or off, depending on the individual bits.

---

[3] You can, of course, use any editor of your own choosing, such as GNU Emacs.

[4] You should remember, from other courses, that the Unix **man** command gives you the manual page for any command. So, for example, you could type "**man pwd**" for more information on the **pwd** command.

[5] If you are at home, you will not have a *~elec2041/cdrom* directory, of course. You *do* have the actual Companion CD-ROM, however. In this case, mount the CD-ROM in some directory (usually */mnt/cdrom*) and use that directory instead.

[6] This manual uses the standard C notation for hexadecimal numbers. You should remember this notation from other courses. You can always refresh your memory by reading Brian Kernighan and Dennis Ritchie's classic book *The C Programming Language*, Second Edition (ISBN 0-13-110362-8).

Task 1: LED Bit Patterns

Your first simple task is to work out the bit patterns (ie, the binary ones and zeros) that will turn all of the eight LEDs on or off. Does a 1 bit turn the LED on or does it turn it off? You may need to consult the *Hardware Reference Manual*.

What value, in binary, will turn the eight LEDs on? What value will turn them off? How did you come up with these values? The answer "I read ahead and found it in the source code" will not be valid!

```
Checkpoint 1: ...................................................................  Signature: ...............................
```

Program Design

After specifying what the program will do, the usual next step is to come up with a pseudo-code or flow-chart version. Computer programming is an *art:* you are meant to be creative. Pseudo-code, flow-charts and the like can help your creativity by providing a high-level view of what you are trying to achieve. As this experiment is only an introduction, most of the hard work has been done for you. So, Figure 4 gives you the pseudo-code for the Flashing LEDs program:

```
program main

const
  Value1   = XXX          /* Value to turn LEDs on */
  Value2   = XXX          /* Value to turn LEDs off */

  LED_port = 0x10000000   /* Address of the eight LEDs */

do {
    LED_port := Value1
    LED_port := Value2
} forever
```

Figure 4: Pseudo-code for the Flashing Lights Program

Replace the *XXX*'s with the values you determined previously, in Task 1.

Program Implementation

Now that you know what is required of the program and have figured out how the program is going to do it (using the pseudo-code above), it is time to implement.

As mentioned previously, the DSLMU Microcontroller Board is based on the ARM 32-bit RISC architecture. In other words, the microcontroller on the board is programmed using the ARM instruction set.[7] In this part of the experiment, you will be programming in assembly language—which an assembler will convert into the binary ones and zeros needed by the microcontroller.

Since this is your first ARM assembly language exercise, Figure 5 gives you the complete program without further ado:

---

[7] For those who are interested, the particular microcontroller on the DSLMU Microcontroller Board, the Atmel AT91R40008, uses the ARM7TDMI core to implement Version 4T of the ARM instruction set. All this might seem confusing, but you don't need to worry about it. See the *ARM Architecture Reference Manual*, pages v–ix, in the *reference* directory of your CDROM, if you want more information.

```
; ***********************************************************************
; *                                                                     *
; *            Flash the LEDs on the DSLMU Microcontroller Board        *
; *                            First Version                            *
; *                                                                     *
; ***********************************************************************
;
; Author:   John Zaitseff <J.Zaitseff@unsw.edu.au>
; Date:     7th March, 2003
; Version:  1.6
;
; This program, when run on the DSLMU Microcontroller Board, flashes the
; LEDs on and off.  A high-level (pseudo-code) version of the program is:
;
;     program main
;
;     const
;       Value1   = 0b11111111   /* Value to turn LEDs on */
;       Value2   = 0b00000000   /* Value to turn LEDs off */
;
;     address
;       LED_port = 0x10000000   /* Address of the eight LEDs */
;
;     do {
;         LED_port := Value1
;         LED_port := Value2
;     } forever
;
; where Value1 is the value to turn ON all of the LEDs and Value2 is the
; value to turn OFF all of the LEDs.


; ------------------------------------------------------------------------
; Constant values used in this program

        .equ    LED_port, 0x10000000   ; Location of LED port

        .equ    Value1,   0b11111111   ; Value to turn LEDs on
        .equ    Value2,   0b00000000   ; Value to turn LEDs off


; ------------------------------------------------------------------------
; Assembly-language preamble

        .text                          ; Executable code follows
_start: .global _start                 ; "_start" is required by the linker
        .global main                   ; "main" is our main program

        b       main
; ------------------------------------------------------------------------
; Start of the main program

main:                                  ; Entry to the function "main"

        ; Although "main" is technically a function, this particular
        ; function has an infinite loop and so never returns to its caller.

        mov     r4, #LED_port          ; Load the address of the LED port
                                       ; (ie, the value of the constant
                                       ; LED_port) into register R4

main_loop:                             ; Start of the infinite loop

        ; Turn the LEDs on

        mov     r5, #Value1            ; Load the value Value1 into R5
        strb    r5, [r4]              ; and store this value (a byte) to
                                       ; the address in R4 (the LED port)
```

(Continued on the next page...)

```
        ; Turn the LEDs off

        mov     r5, #Value2          ; Load the value Value2 into R5
        strb    r5, [r4]             ; and store this value (a byte)

        b       main_loop            ; Branch back to main_loop, ie, run
                                     ; forever
; --------------------------------------------------------------------
        .end
```

Figure 5: Program *flash-v1.s*

Before you proceed any further, take the time to read through this program carefully and see if you can figure out what is going on. The following points may be useful:

- Anything following a semicolon "`;`" is a comment that is ignored by the assembler,
- A name (a sequence of letters, digits, "`_`" and/or "`.`") followed by a colon "`:`" is a *label;* labels are used to refer to a particular point in the program from anywhere else,
- Anything starting with a full-stop and *not* followed by a colon, such as `.equ` and `.text`, is an *assembler directive;* assembler directives, naturally enough, tell the assembler what to do,
- Anything else, such as "`strb r5, [r4]`" is an assembly language statement, also called an *instruction*.
- Almost all ARM assembly language instructions have the form "`mnemonic parameters`". Most of those statements having two parameters have the form "`mnemonic destination, source`". It is unfortunate, however, that some instructions—including `strb` ("store byte") in this program—have the reverse form "`mnemonic source, [destination]`"!
- Numbers of the form `0x10000000` are, of course, in hexadecimal.
- Numbers of the form `0b11001100` are in binary. Ordinary numbers are in decimal.

Programming Style

A few comments may be appropriate at this point about the style of writing assembly language programs. Programmers tend to be an individualistic group of people, and nowhere is this more evident than in how they choose to format their code. "Holy wars"[8] have even erupted over trivial things like the position of the brace character "{"!

Nevertheless, it is important for readability and consistency to choose an appropriate style of writing programs and then stick to it! And whilst programmers that have decades' worth of experience will usually refuse to change their style, you are most likely still forming yours…

For this reason, you are encouraged to adopt the following style rules, especially for this course:

- Start each source code file with a heading,
- Include your name and e-mail address so that it is easy to see who is responsible for the file,
- Display the date of last modification and a version number,[9]
- Have a description of what the program does, possibly with the pseudo-code version included,
- Include *appropriate* comments that explain the "why" and not just the "how" of the program throughout the source code, and

---

[8] See the excellent *Jargon File* on the Web at *http://www.catb.org/~esr/jargon/html/* for more information on this and many other computer-related topics.
[9] Unless, of course, you are using a version-control system, which maintains this information for you.

- Use a sensible layout for your code. This makes it easy to see the assembly language statements, and makes any labels in your code easy to see.

You can use the program *flash-v1.s* in Figure 5, as an example of what to do (be aware, however, that some of the comments are *not* particularly appropriate once you understand the ARM instruction set). Experiment 2 will cover the topic of programming style in more detail.

Please note that programs shown in later experiments have had many of their comments removed to save paper. This does *not* mean you can do the same! You should always consult the actual source-code files on your CD-ROM for the full listing.

Text Editing

Theory is all well and good, but what about the practical aspects of actually running the program? To work!

Since this experiment is not about learning to type, you can save yourself a lot of time by using an electronic copy of the source code. The appropriate file is already in your working directory and is called *flash-v1.s*—the same name as used in Figure 5.

You now need to open this file for editing. There are two ways to do this: the "traditional" shell method, and the graphical method; you can choose to use either one.

The first method is to issue the appropriate command line in the shell window. It has the advantage of being quick and easy (no fiddling around with Open File dialog boxes and so on). Simply type:

```
kate flash-v1.s &
```

The second method is to click on the **Editor** icon in the top left-hand corner of the screen. Once the editor starts, select **File » Open**… from the menu, then navigate your directory tree to *exp1* in the Open File dialog box before (finally!) selecting the file *flash-v1.s* and clicking OK.

Irrespective of the method you choose to use, the first thing you need to do is add your name and your partner's name to the list of authors. Save the changes by selecting **File » Save** from the menu. Don't quit the editor: you have only just begun!

Assembling the Program

Development in the real world tends to be *iterative*, as illustrated in the following flow-chart:
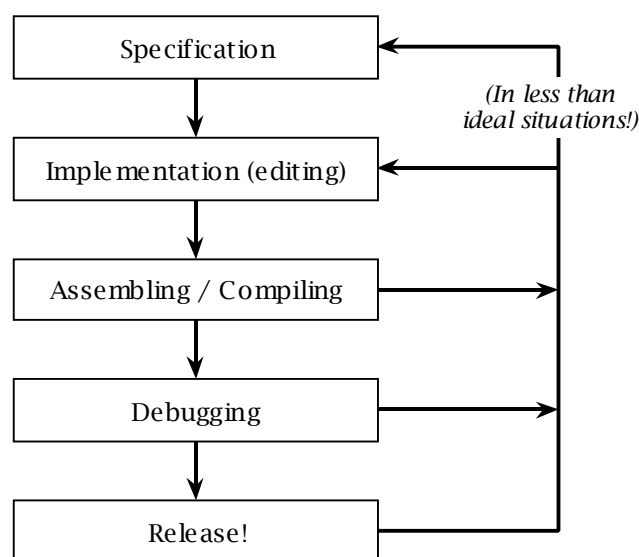
Figure 6: Typical Application Development Flow-chart

Now that you have done the Editing phase (you will be doing this by yourself from Experiment 2 onwards, of course), you need to assemble the program. This is a two-step process. The first step is to *assemble* the source code file into a relocatable object file, using the assembler **arm-elf-as**. The second is to *link* that object file into a final executable, using the linker **arm-elf-ld**.[10]

Use the mouse (or ALT+TAB) to select the shell window once again and type the following commands. The first line corresponds to the first step; the second to the second step:

```
arm-elf-as --gdwarf2 -marm7tdmi -o flash-v1.o flash-v1.s
arm-elf-ld -o flash-v1.elf flash-v1.o
```

By the way, Linux (and Unix in general) tends to be the "strong, silent type": if everything went well, no messages will appear in the shell window. If either the assembler or the linker complain, check the command lines carefully and make sure the file you are trying to assemble is the correct one. If you are still experiencing troubles, call a demonstrator!

Why bother with a two-step process? As you will see from later experiments, you can combine any number of *relocatable object files* to form an *executable*. "Relocatable" simply means that the final location of code and data has not been fixed into place, ie, the code and data can be linked to any particular addresses at a later stage.

If you now list your directory (using the **ls** command), you will find the following files, amongst others:

flash-v1.s	– the source code for the program,
flash-v1.o	– the relocatable object file (the result of **arm-elf-as**), and
flash-v1.elf	– the actual executable file (the result of **arm-elf-ld**).

The technical relationship between these files is called *dependence*. In particular, *flash-v1.elf* depends on *flash-v1.o*; *flash-v1.o* in turn depends on *flash-v1.s*. This means that if *flash-v1.s* is modified, all files that *depend* upon it (*flash-v1.o* and *flash-v1.elf*) need to be recreated (using the same commands as above). Although this point is obvious, it is important to note it!

By the way, later on you will be using a tool called **make** to help you maintain dependence relationships. Of course, there is nothing to stop you from using these tools now, if you are comfortable in doing so![11]

Running the Program

Now for the smoke test, as electrical engineers are wont to say! To run the program, you need to:

1. connect to the DSLMU Microcontroller Board,
2. transfer (download) the program to the board, and
3. instruct the board to run the program.

You should still be in the Shell window. Type:

```
kmd &
```

The Komodo debugger should automatically connect to the DSLMU Microcontroller Board, then put up the window shown in Figure 7:

---

[10] The **arm-elf-** prefix simply shows that these tools produce output for the ARM architecture.

[11] In this course, each part of an experiment is supplied with its own *make-file*, a "script" that tells **make** what to do. The relevant make-file is usually the name of the source code file, but with a *.make* extension instead of *.s* or *.c*. In this case, the make-file for *flash-v1.s* is *flash-v1.make*. Simply type "**make -f flash-v1.make**" to recreate *flash-v1.elf* every time *flash-v1.s* is changed.
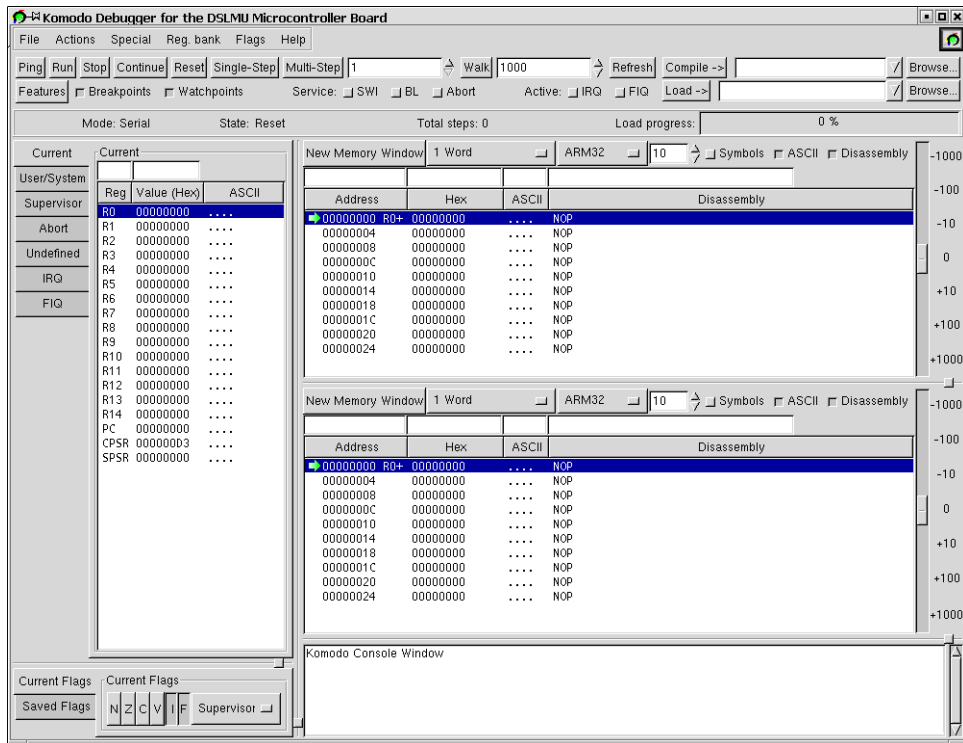
Figure 7: The Komodo Debugger

If it does not, check that the cable between the host PC and the board is firmly plugged in and that you did not make any mistakes in typing the command line. Try again; if it still fails, ask a demonstrator for help.

You now need to download the program to the board. Click on the *second* **Browse** button, in the top right-hand corner (ie, the one associated with the **Load** button, *not* the **Compile** button). Now, select *flash-v1.elf* from the Select Object File dialog box. Click **OK**, then click **Load** in the main window. You should see the Load Progress indicator move to 100%.

Now that your executable has been downloaded to the DSLMU Microcontroller Board, you need to prepare Komodo to run it. The first step in doing so is to press the **Reset** button (in the top left-hand corner of the Komodo window).

Next, set the program counter to the start address of the program: this is the value of the symbol **_start**. Use the mouse to click on the PC (program counter) register, on the left-hand side of the window. Select the *second* edit box above the list of registers (just underneath and to the right of the word "Current") by clicking the left mouse button in it. Delete everything in this edit box (by using the BACKSPACE and arrow keys), then type in the label **_start** and press ENTER. The list of registers will be updated to reflect the new value, as shown in Figure 8:

Figure 8: The Registers

You are almost there… Find the edit field just above the top-most **Address** label, click your left mouse button in it, type **pc** and press ENTER. Also click the **Symbols** checkbox in the same area. You should see something like the screenshot in Figure 9:
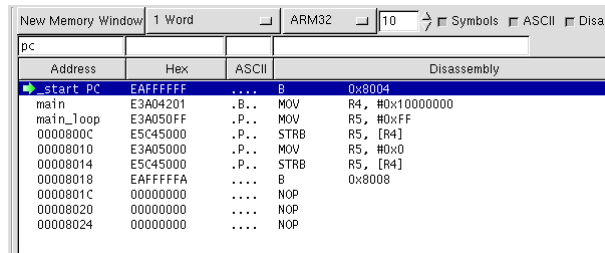

Figure 9: The Memory Panel

Finally, click the **Run** button, in the top left-hand corner of the Komodo window.

If everything is successful, you should see the eight LEDs turn on. But they don't flash on and off! Why not?

Before continuing, you need to stop the program. To do this, press the **Stop** button, next to **Run**.

Debugging Your Program

There are, essentially, two ways to find out why the program is not working as expected (no, guessing does not count!): you could examine the software, and/or you could examine the hardware. In general, you may need to do both.

A good debugger can help you solve software problems in three main ways: by controlling how the program is run, by examining variables and by allowing you to change the contents of those variables.

The first way the debugger helps you is by letting you control how the program is run. In particular, you can run the program you are debugging up to a particular point (called a *breakpoint*, as the program "breaks" at that point); you can proceed step-by-step (appropriately enough called *stepping*); you can even jump to any place within your program (but very carefully!).

The second way the debugger helps is by allowing you to examine the state of execution: the program's variables, the processor registers, the actual memory contents. In addition, you can halt the program's execution whenever one of these is read or changed—this is called *watching* a variable.

The third way the debugger helps is an extension to the second: it allows you to actually change the state of the program's execution. In other words, it allows you to change the contents of variables in the processor registers and in the actual memory.

These concepts might sound complicated, but in practice they are quite easy to understand. As a first pass, you will step through the entire program, trying to understand what is going on. However, you can appreciate the fact that this is simply not feasible for large programs!

Task: Before you start stepping through your program, you need to restart it. Press the **Reset** button in the Komodo window, select the PC register as before, change the value from 00000000 to `_start`, then press ENTER.

Now you can run the program step-by-step. Press the **Single-Step** button *once*. This will execute the instruction being pointed to by the green arrow and update the display. The arrow will now point to the instruction that will be executed next.

Two important points to mention:

- Please ignore the blue highlight—it has a rather obscure meaning. It is the *green arrow* that you should look at. It so happens that the blue highlight and the green arrow point to the same place (for the moment), but that is not always the case.

- The line of code pointed to by the green arrow has *not* been executed yet: it is the one that will be executed *next*. This is easy to forget! And this is one of the reasons why it is helpful to keep the editor window (with the program's source code) always open.

Task 2: Stepping Through the Program

Use the **Single-Step** button to go through your program step by step. What happens to the LEDs as you do so? What registers change their values, and at what points in the program does this happen? Be ready to explain what is happening to the Laboratory assessor.

> Checkpoint 2: .... Signature: ....

Slowing It Down

You have probably figured out by now that the reason *flash-v1.s* does not seem to work is that the LEDs simply turn on and off at an extremely fast rate—far too high for you to observe directly. What to do?

The solution, of course, is to slow down the rate of flashing. The simplest way to do this is by adding *delay loops:* bits of code that don't really do anything, except waste time. You will look at this topic in greater depth in Experiment 4; for now, use the Editor to open the file *flash-v2.s*. Carefully examine this new program, reproduced in Figure 10 below:

```
; **********************************************************************
; *                                                                    *
; *           Flash the LEDs on the DSLMU Microcontroller Board        *
; *                          Second Version                            *
; *                                                                    *
; **********************************************************************
;
; Author:   John Zaitseff <J.Zaitseff@unsw.edu.au>
; Date:     7th March, 2003
; Version:  1.6

; This program, when run on the DSLMU Microcontroller Board, flashes the
; LEDs on and off.  A high-level (pseudo-code) version of the program is:

;     program main

;     const
;       Value1   = 0b11111111   /* Value to turn LEDs on */
;       Value2   = 0b00000000   /* Value to turn LEDs off */
;       WaitVal  = 10000        /* Number of loops to wait */

;     address
;       LED_port = 0x10000000   /* Address of the eight LEDs */

;     do {
;         LED_port := Value1
;         delay(WaitVal)
;         LED_port := Value2
;         delay(WaitVal)
;     } forever
```

(Continued on the next page…)

```
;       function delay (integer DelayVal)

;          while DelayVal > 0
;              DelayVal := DelayVal - 1

; where Value1 is the value to turn ON all of the LEDs, Value2 is the
; value to turn OFF all of the LEDs and WaitVal is the number of loops
; to repeat whilst waiting.
; ------------------------------------------------------------------------
; Constant values used in this program

        .equ    LED_port,       0x10000000      ; Location of LED port
        .equ    Value1,         0b11111111      ; Value to turn LEDs on
        .equ    Value2,         0b00000000      ; Value to turn LEDs off
        .equ    WaitVal,        10000           ; Number of loops to wait
; ------------------------------------------------------------------------
; Assembly-language preamble

        .text                           ; Executable code follows

_start: .global _start                  ; "_start" is required by the linker
        .global main                    ; "main" is our main program

        b       main
; ------------------------------------------------------------------------
; Start of the main program

main:                                   ; Entry to the function "main"

        ; Although "main" is technically a function, this particular
        ; function has an infinite loop and so never returns to its caller.

        ldr     r4, =LED_port           ; Load the address of the LED port
                                        ; (ie, the value of the constant
                                        ; LED_port) into register R4

main_loop:                              ; Start of the infinite loop

        ; Turn the LEDs on

        mov     r5, #Value1             ; Load the value Value1 into R5
        strb    r5, [r4]                ; and store this value (a byte) to
                                        ; the address in R4 (the LED port)

        ; Wait for the specified period of time

        ldr     r0, =WaitVal            ; Pass the parameter, WaitVal, in R0
        bl      delay                   ; Call the function "delay"

        ; Turn the LEDs off

        mov     r5, #Value2             ; Load the value Value2 into R5
        strb    r5, [r4]                ; and store this value (a byte)

        ; Wait again for the specified period of time

        ldr     r0, =WaitVal            ; Load the parameter into R0
        bl      delay                   ; Call the function

        b       main_loop               ; Branch back to main_loop, ie, run
                                        ; forever
; ------------------------------------------------------------------------
; Function "delay": delay program execution by wasting time in a loop

delay:

        ; This function expects R0 to contain the number of loops for
        ; which to wait.  It does not return any meaningful values.  The
        ; value in R0 is destroyed.  All other registers are preserved.
```

```
delay_1:
        subs    r0, r0, #1               ; Decrement the number of loops to go
        bne     delay_1                  ; and repeat if it is not yet zero

        mov     pc,lr                    ; Return to the caller

; ------------------------------------------------------------------

        .end
```

Figure 10: Program *flash-v2.s*

You should study this program in detail, trying to understand just what is happening. Hopefully the comments in the program will be sufficient for you to do so!

Task: Before you can proceed with running a new program, you need to finish off with the old one. To do this, press the **Stop** button in Komodo, then select **File » Quit Program** (in other words, select the **File** menu with the left mouse button, and, while holding the mouse button down, select **Quit Program**). You will be returned to the Shell window.

Once you have examined *flash-v2.s*, you need to assemble and link it to create the executable *flash-v2.elf*. You can do this by typing in the following command lines at the shell prompt:

```
arm-elf-as --gdwarf2 -marm7tdmi -o flash-v2.o flash-v2.s
arm-elf-ld -o flash-v2.elf flash-v2.o
```

You will find that typing in long command lines again and again will get *very* tedious! The alternative is to use the **make** command. Learning how this command works is beyond the scope of this experiment, but you *can* use it without knowing much about it![12] To use **make** to recompile *flash-v2.s*, type:

```
make -f flash-v2.make
```

This is shorter than typing out the two `arm-elf-` command lines, but it is still a little long... You can make it even shorter, but that is left "as an exercise for the reader".[13]

Task 3: Multiple Steps

Assemble and link the program *flash-v2.s*. Use Komodo debugger to run your program. Once you verify that the LEDs actually flash, go through your program step by step. What do you discover about the feasibility of doing this? How many steps would you have to take to step through *one* cycle of the program (ie, to turn the LEDs on, then turn them off)? Can you figure out how to overcome this problem by using the **Multi-Step** button?

Challenge: The program *flash-v2.s* has a (very small and deliberate) problem with it. Can you find it? Hint: you will not see this problem ("bug") by merely running the program! (The moral of this challenge is that testing and debugging, as essential as they are, do *not* guarantee a correct program).

Be ready to explain what is happening to the Laboratory assessor.

Checkpoint 3: .... Signature: ....

---

[12] If you plan to earn your living as a computer programmer or engineer, it is probably very much worth your while to learn how to use **make** effectively. The *GNU Make Reference Manual* is not a bad place to start reading; you can find this Manual in the *gnutools/doc* directory on your CD-ROM.

[13] The **make** command looks for the file *Makefile* by default; specifying "**-f** *make-file*" on the command line (for some file *make-file*) overrides this. If you *did* have the file *Makefile*, all you would have to type is "**make**": just four letters! If you like, you can use **mv** to rename the file *All.make* in your directory to *Makefile*. Then, every time you type "**make**", *every* program in that directory will be made.

A Program in C

Most programmers in the real world do not use assembly language to write their programs. Instead, they prefer to use *high-level languages* like C, C++ or Pascal. The reason for this is that assembly language programs are too tedious to write and, more importantly, maintain. And yet it is **very** important for you to know how software works at its most fundamental level, at the level of assembly language. That is why *you* are studying this course.

Among other things, this course is designed to teach you how programs written in those high-level languages are actually implemented when run on real hardware. Later experiments will deal with this topic in greater detail; this first experiment will just give you a foretaste of what is to come.

Given the above, the following figures show what a "real world" programmer might write if asked to implement a program to flash the LEDs. Figure 11 shows the contents of *flash.h*,[14] a file containing definitions. Figure 12 shows the actual program, *flash.c:*

```
/* Type definitions */

typedef unsigned char byte;                    /* 8-bit byte */

/* Constant declarations */

#define FALSE (0)                              /* C value for FALSE */
#define TRUE  (! FALSE)                        /* C value for TRUE */

#define LED_port_addr (0x10000000)             /* Address of LED port */
#define Value1  0xFF                           /* Value to turn LEDs on */
#define Value2  0x00                           /* Value to turn LEDs off */
#define WaitVal 10000                          /* Number of loops to wait */

/* Function prototypes */

int main (void);
void delay (int delayval);
```

Figure 11: Program Header *flash.h*

```
/**********************************************************************
*                                                                    *
*         Flash the LEDs on the DSLMU Microcontroller Board          *
*                                                                    *
**********************************************************************/

/*  Author:    John Zaitseff <J.Zaitseff@unsw.edu.au>
    Date:      6th March, 2003
    Version:   1.5

    This program, when run on the DSLMU Microcontroller Board, flashes the
    LEDs on and off.  It does the same thing as the second version of the
    assembly language program.
*/

#include "flash.h"    /* The compiler inserts everything in Figure 11 here */

/*  ------------------------------------------------------------------
    Function:    main            - Main program entry point
    Parameters:  (none)          - No parameters are passed
    Returns:     (nothing)       - This function never returns

    This function performs the main work of the program, flashing the LEDs
    on and off.
*/
```

---

[14] Please note that the listing in Figure 11 has had most of its comments removed. This is so that you can instantly see just what is necessary. When writing your own programs, you should follow the example set by the file on your CD-ROM, *not* by what appears in this figure!

```
int main (void)
{
    byte *LED_port = (byte *) LED_port_addr;    /* LED port */

    while (TRUE) {                               /* Do this forever */
        *LED_port = Value1;                      /* Turn the LEDs on */
        delay(WaitVal);                          /* and wait for WaitVal */

        *LED_port = Value2;                      /* Turn the LEDs off */
        delay(WaitVal);                          /* and again wait */
    }
    return 0;                                     /* We don't return, but */
                                                 /* keep the compiler happy */
}

/*  ------------------------------------------------------------------------
    Function:    delay          - Delay program execution by wasting time
    Parameters:  delayval       - Number of empty loops to repeat
    Returns:     (nothing)      - No value is returned

    This function delays the program by "spinning in an idle loop".  In other
    words, it just repeats a "do nothing" statement "delayval" number of times.
*/
void delay (int delayval)
{
    int i;

    for (i = 0; i < delayval; i++)
        ;                                        /* Do nothing */
}
```

Figure 12: Program *flash.c*

Take the time to compare this program with *flash-v2.s*. Both do the same job. Which do *you* think is easier to understand?

You need to run a *compiler* to convert code written in a high-level language into something that can be run on a real machine. In this case, you need a C compiler. In this Laboratory, you will be using the GNU C Compiler, **arm-elf-gcc**. Just like the assembler **arm-elf-as**, the C compiler requires a two-step process: the first step *compiles* the source code into a relocatable object file, the second *links* that object file into a final executable. The following commands correspond to these two steps; note that the option **-O2** uses the upper-case letter "O", not the number zero:[15]

```
arm-elf-gcc -c -mcpu=arm7tdmi -O2 -g -Wall -o flash.o flash.c
arm-elf-ld -o flash.elf cstart.o flash.o
```

Typing such command lines is definitely error-prone and rather laborious! For this reason, you should just use **make**:

```
make -f flash.make
```

Task 4: Running the C Program

Compile the program *flash.c* and run it on the DSLMU Microcontroller Board using the Komodo debugger. Show your working program to the Laboratory assessor.

Checkpoint 4: .... Signature: ....

---

[15] Some details have been glossed over here. In particular, the file *cstart.o* provides some "glue" that allows the final executable to be run on the DSLMU Microcontroller Board. This file has already been assembled for you; you can look at the original source code in *cstart.s*, if you are interested in such details.

Use the GNU C Compiler to convert the program from C to assembly language. The appropriate command line is:

```
arm-elf-gcc -S -fverbose-asm -O2 -o flash-ctoasm.s flash.c
```

Remember that the option **–O2** uses the upper-case letter "O", *not* the number zero!

Now, use the Editor to open the compiler-generated file, *flash-ctoasm.s*. Try to correlate this code with the original C program in Figure 11 and Figure 12. In addition, compare the compiler-generated code with the hand-written code in *flash-v2.s* (Figure 10). What differences do you note? Are there any similarities? Can you determine the purpose of the registers being used in *flash-ctoasm.s*? Give a brief explanation of this file to the Laboratory assessor.

Caution: the GNU Assembler, **arm-elf-as**, allows you to use both ";" and "@" to start a comment. These experiments use ";", as that is what code written by ARM programmers tends to use. The GNU Compiler, however, uses "@". All you have to do is remember to treat "@" the same way you treat ";".

Checkpoint 5: .... Signature: ....

Saving Your Work

You have almost finished this experiment! The only thing left to do is to save your files to a floppy disk, so that you can use them at home or the next time you come into the Laboratory. You can do this under Linux by using the **mtools** package. This package provides commands that are very similar to those under MS-DOS.

Insert your 3½˝ floppy disk into the drive and type:

```
mdir a:
```

This lists everything in the root directory of the floppy disk. Hopefully, your floppy disk will be blank or, at least, have enough room on it to store all of your files. If not, you can always type "**mformat a:**" to format the floppy disk. **Caution: this will delete everything from the floppy!**

Now, create a subdirectory in which you will store your work files:

```
mmd a:exp1
```

Finally, copy all of the files in your current directory to the *exp1* directory on the floppy:[16]

```
mcopy * a:exp1
```

When you want to restore the files from the floppy disk, you use **mcopy** again, "in reverse". For example, to copy everything from the *exp1* directory on the floppy disk to the current directory, type:

```
mcopy a:exp1/* .
```

Please note the "**.**" at the end of this command. In addition, **mcopy** uses "**/**" instead of the MS-DOS "****". If you need more information on how **mcopy** works, please consult the on-line documentation; you can access that documentation by typing "**man mcopy**".

---

[16] This command will copy *everything* in the current directory to the floppy disk. However, there is usually no need to save the *.o* object files or the *.elf* executable files. You can use standard Unix wildcards or specific names instead of "*****". For example, "**mcopy *.s *.c *.h *.make a:exp1**" should copy the relevant files for this experiment.