ECE352F: Computer Organization

Lab #1: Standalone C Programming

Fall 1998

1 Preparation

- Read through the lab and do the preparation indicated.

2 Introduction

In this lab you will do some exercises to demonstrate:

- features of the gnu compiler
- the use of nm to look at the symbol table when debugging
- techniques for programming the gizmo board

All page references will be for Version 2.2 of the Gizmo manual.

Copy the directory tree starting at *~pc/352/lab1* on the ugsparcs to your account.

`ugsparc1%` **cp -rp ~pc/352/lab1 .**

All of the exercises will be found in subdirectories of this directory.

3 A Simple Compile

Change into the directory called **sect1**. Here you will find a C program called *hello.c* and a *makefile*.

Examine the contents of *hello.c*. Note that it contains the standard first program that everyone uses to test their C programming skills! Examine the include file that this program uses:

`ugsparc1%` **more ../include/StandAlone.h**

This file contains a number of declarations for routines that are declared in a library of standalone C routines. The most important of these routines help you to do some terminal I/O and to peek and poke at memory. The source for these routines can be found in the files in the *../lib* directory, and you link in the *../lib/libSA.a* module to access the routines. The print routine used here has most of the same features as the standard Unix *printf*. Note that the linker will load all procedures in a module (module.o), even though you may only use one of the procedures. If you are concerned about memory space, then it is important to be aware of this.

Examine the makefile. When you type `make hello`, this will use the gcc compiler to compile *hello.c* and *crt0.s*. This makefile explicitly runs the linker so that you can include your own *crt0.o* routine, which is the C startup routine.

Look at *crt0.s*. Note that it has a label called *_start*, which corresponds to the symbol after the "-e" flag in the "ld" phase of the makefile for *hello*. This will be where your program starts, otherwise known as the entry point. The specification of "-Ttext 30000", tells the linker that the first instruction in the instruction segment, or text segment, is to be placed at address $30000. This is usually the same as the entry point of your program, but it need not be. This linker also has a "-Tdata" flag that is used to specify where the data is to be loaded. This is important when you want to have your program in ROM, and the data in a different address space corresponding to RAM. Without specifying the data address, the data will be loaded immediately after the text segment.

The first thing that *crt0.s* does is to initialize the stack. Why is this a good idea? It then jumps to a subroutine called *_startup*. This subroutine can be found in the lib directory:

`ugsparc1%` **more ../lib/startup.c**

It is a procedure written in C. Note that here, the procedure is called just *startup*, without the underscore ('_'). This is a convention in this compiler, where all symbols in a C program have an underscore prepended to them when they appear in an assembly language program. Knowing this will help us to link C and assembly routines together.

There are two external symbols declared, *end* and *edata*. They are symbols generated by the compiler. By convention, a C program has two types of data. One is *initialized* data, which is data that already has an initial value when the program is loaded. For example, the statement:

```
int done = 1;
```

declares the variable *done* as an integer that should start with the value of 1. The other data type is called *uninitialized* data, otherwise known as data in the *bss* segment. For example:

```
int varname;
```

declares the uninitialized variable *varname*.

By convention, variables in the bss segment are initialized to zero (0) before the program starts. There are C programs that will not work unless the bss segment is in fact initialized to zeroes, even though proper programming style would say that this variable is uninitialized, and you have no right to assume it has any value. The symbol *edata* is the first address of the bss segment, and the first address location after the data segment. The *end* symbol is the last address in the bss segment. The *bzero* procedure called in *startup* will handle initializing the bss segment.

Once the initialization is complete, the main routine of your program is called, which actually starts the program that you wrote. When your program exits, it needs to go somewhere, preferably the monitor. In *crt0.s*, there is a `trap #15` instruction, that is what you would normally do in a simple assembly language program to get back to the gizmo monitor. For reasons that no one has bothered to chase, this does not provide a clean exit from your program. It is better to call *_exit()* at the end of your main program. This is described on Page 43 in the Gizmo Manual.

Now compile your program:

`ugsparc1%` **make hello**

and test it out by downloading the resulting executable file to your board. We can use the ethernet connection for a quick download. You will need to know the internet name of your board. It should be labeled near the ethernet cable connector. If this download procedure fails to work, try resetting your board first.

`ugsparc1%` **tftp uggizmoNN hello**

You will get the message

```
Hit ^A when Unix prompt appears.
```

If you are issuing the *tftp* command through the auxillary port of the the gizmo board, then you will need to do this. If you issued the *tftp* command through another network window, then ignore it.

When your program is loaded, it will immediately execute. Remember that the starting address is at $30000. You should be able to rerun your program using the monitor command:

`>` **30000:j**

if you called *_exit()*.

To see how a *fully-automated* compile works, examine the makefile again. The entry for *hello2* shows how to invoke gcc without invoking the loader explicitly. Also, look at the source file. You will see that it now uses *printf* and includes *stdio.h* as you would in a regular C program. Many of the standard C library routines are available to you on the gizmo board, and except for the *_exit()* glitch, this program would compile on any system. The standalone library used for *hello.c* is provided for your use and inspection.

Now compile your program again using the *fully-automated* compile:

`ugsparc1%` **make hello2**

You will get a lot of messages printed by the compiler. This is because the "-v", for verbose, flag was used. For a quieter compile, you can delete this flag. With the flag, you can see the various activities of the compiler. Look for the place where the linker is invoked. This is a different linker than the one used previously. It will eventually invoke the one we have been manually calling. This has to do with the way *gcc* is set up. Note that the linker picks its own *crt0.o* module to link with. This one is much more involved because it handles more of the devices on the board and the standard I/O conventions of C.

Test out this version of the program. At what address is the text segment loaded?

3.1 Preparation and Exercise

1. The *print* routine used in *hello.c* is a simplified version of the standard *printf* routine used in the C standard I/O library. Go to the *lib* directory and trace the sequence of procedure calls that are made until a character is output on the screen. What are the names of the procedures? (**Hint:** You may find the Unix *grep* command useful.)
2. Write a much simpler C routine called **myprint(char *)** that takes a string as an argument and prints it on the terminal. You should call a procedure called **myputch(char)** that takes a character and outputs it on the terminal using polled I/O. This would be the equivalent to what you have done in the past in assembly language.
3. Test your program in the lab.

4 Using nm

In this section, we will use a program called *nm*. The version that we want to run is actually `/gnu/gizmo/bin/nm`, not the normal system version. To make your life easier, you might want to set up the following alias:

`ugsparc1%` **alias nm68 /gnu/gizmo/bin/nm**

When using a compiler, you do not generally have the convenience of an assembly listing to tell you at what addresses various procedures, labels, or other symbols, are located. If you wish, for example, to set a breakpoint in a program at the start of a particular procedure, then you will need to know the starting address of that procedure. This is also true for variables. When you want to examine their contents, you will need to know at what address they are stored. This is done by examining the symbol table, which is part of the object file. In your **sect1** directory, where you compiled your *hello* program, type the following:

`ugsparc1%` **nm68 -n hello | more**

The fully linked program is in the file *hello*. The flag, "-n", orders the output by address. Look for the variables and procedures in your program and identify whether they are in the text segment ("T" or "t"), the initialized data segment ("D") or the uninitialized segment ("B"). Look for the symbols *etext*, *edata*, and *end*. You will also see a lot of other symbols and routines that are generated by the compiler. If you want to know more, the man page is is a good place to start.

Load *hello* again, and examine the memory locations for *globalvar* and *initglobalvar*. What are their values? Can you find localvar? It is a local variable, and thus only appears on the stack. Therefore, you will not be able to find it in the symbol table.

Use the monitor to disassemble the first few instructions in memory at the starting point of the program. Confirm that thse instructions correspond to the instructions in *crt0.s*.

5 Using Interrupts from C

Embedded programs almost always involve the use of interrupts. In this part you will examine a simple program that shows how to invoke an interrupt service routine that is written in C.

The program will use the autovector 6 interrupt for DUART 0. Using vectored interrupts involves the ICU, and we will not get into that here! See Page 75 for more details.

The main routine of the program prints out the alphabet continuously. Whenever a key is hit on the keyboard, an interrupt is generated. If the key was a "q", then the program will terminate. Otherwise, the interrupt service routine will print out two rows of numbers.

Change into the **sect4** directory for this lab and examine the program there called *alphanum.c*.

C pointers are the best way to access the DUART registers. This program shows two variations. If there are a number of registers, it is often more convenient to declare a structure. Here, the structure called *duart* shows how to declare the first half of the registers that represent the "A" registers. Just add more entries in the structure to get the rest of the registers. Note that because each register is only eight bits, we use a char declaration. If you have to worry about loading registers with unsigned (positive numbers only) or signed (sign extension happens) numbers, then you may or may not need the *unsigned* qualifier. Also, because the registers are on odd address boundaries (see Table 4 in manual), then we add the dummy declarations to put in the proper spacing and alignment. Later in the code, you will see a pointer called *terminal* assigned to the address of DUART0. Now, when you reference elements in the structure, you access the appropriate register.

You can also assign a pointer to a specific address location. There is a *char* pointer called *duimr* that is initialized to point at the mask register. Note that the constant *DUIMR* comes out of the *mapgizmo.h* file in the include directory. It is the address of the mask register.

When declaring these pointers, it is important to declare the correct data size for your pointers and the correct data type!

The setvector procedure is used to load the address of the interrupt routine at the interrupt vector location. It takes as arguments the vector number and the address of the interrupt routine, which in this case is called *intrstub*. To find this routine, you will have to look in the file *intrstub.s*. Look at this file. It provides the wrapping that allows a C procedure to be used as the interrupt service routine. An interrupt routine needs to save all the registers and use the *rte* instruction instead of *rts* when it is complete. The C routine can be called from within this routine. In this case, we need to provide a C routine called *interrupt*, which you will find back in the source for *alphanum.c*.

The *spl0()* procedure sets the processor priority to zero (0). You can find this procedure defined in *crt0.s*, which is much simplified from the actual *crt0.s* being used. C procedures can return a value, and in this case, the current value of the status register is returned in d0, which has been defined in this compiler as the register for return values. The other instruction loads the status register with the new processor priority.

In the interrupt routine, there is another *spl0()* call. When an interrupt occurs on the 68K, the processor priority is set to the level of the acknowledged interrupt. What does this call to *spl0()* do for you? What happens if you remove it?

Compile, load, and try the program. Also try the program without the call to *spl0()* in the interrupt service routine.

6 Initialized Variables Again

Modify *alphanum.c* so that the variable *done* is initialized in the declaration

```
int done = 0;
```

instead of using a statement in the program.

```
done = 0;
```

Compile and test your program. Type "q" to end the program and then start your program again, without reloading it. What happens? Reload your program from the host and run it. Explain what is going on.

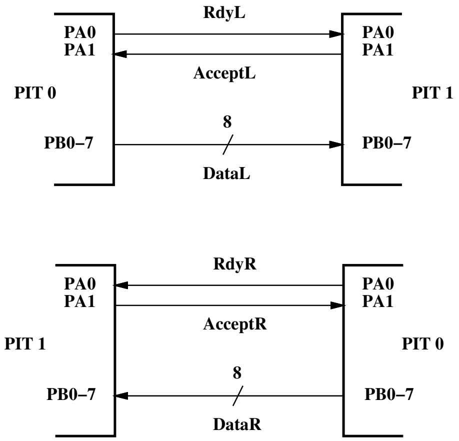
7 Gizmo Communication

In this exercise, most of your programming should be done in C.

By using the PITs, set up a communication link between two gizmo boards. To make your life easier, do not use interrupts. Initially, you can set up a unidirectional link using a handshaking protocol similar to that shown in Figure 4.20 of the text.

7.1 Preparation and Exercise

1. Write the C program to implement the communications link. Start with the unidirectional link. A suggested connection scheme is shown in the figure below. This will allow you to connect to other boards running different implementations of the program.



Transmit on PIT 0
Receive on PIT 1

Note: There is no guarantee as to what state the gizmo board monitor has initialized the peripheral chips. You should make sure that the PIT is in the correct state. A reasonable PIT initialization is shown on Page 83 of the manual.

2. Devise a scheme to debug your program without using the Gizmo board.

   Hint: Obviously, you will not be able to debug whether you have configured the PITs properly, or whether you are accessing the PIT registers correctly. However, you should be able to decouple the code that has to do with the *protocol* of the link from the code that handles the physical transfer of the bits. Make sure you debug the protocol first before coming to the lab.

3. In the lab, two boards will be required, so you will need to work with another team.

   What is the throughput through your link, measured in *bytes/second*?

4. Once you have the link working, make the link bidirectional. Do not use separate 8-bit connections. What is the throughput in this link, assuming that there is data flowing in both directions?

5. If you still have time, try using interrupts.