# Lab #1: Standalone C Programming

ECE352F: Computer Organization

Fall 1999, Rev. 1

## 1 Preparation

- You should scan through some of the new Ultragizmo lab manual to get familiar with the new monitor. In particular, look through Sections 1, 2, 6, and 7. The material in 6.5.1 and 6.5.2 is incorrect. Scan Section 8 just to see what is there. You may need some of it later.
- 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 and the disassembler, which are useful for debugging
- techniques for programming the gizmo board

All page references will be for Version 3.0 of the Ultragizmo 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*. You will see that it 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 Ultragizmo board.

Examine the contents of *hello.c*. Note that it contains the standard first program that everyone uses to test their C programming skills! It looks like any C program. You can compile it on the ugsparcs and it will work.



If you now want to cross-compile it for the Ultragizmo, i.e., using a compiler on the ugsparcs to create a file that will run on the 68000 processor, what things have to be considered? For example, what about terminal input and output? You might want to poke around in the directory

`/local/src/gizmo/lib/libgizmo2`

Can you find where the DUART address is specified?

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 and the gizmo2 (Ultragizmo) specific library routines. The file *crt0.S* is the C startup routine, which is discussed below.

Note that the linker will load all procedures in an object 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.

We are using a version of the gcc compiler that uses *coff* which stands for *common object file format*. This was a standard developed for the format of the object files. This would make it easier for various tools and utilities to be compatible across many platforms. Alas, the latest standard is called *ELF*, but we could only find this *coff* compiler, so we need different versions of the utilities.

At this time you should look at the first part of Section 6.5 (page 81) in the manual about how symbol names translate between a C program and the assembly language program. In the compiler that we are using a different convention is used. The assembly language programs are written in an *assembler-independent* fashion. A C variable, which might be declared as

```
int CvariableName;
```

is represented as

```
SYM (CvariableName)
```

in the assembly language file instead of

```
_CvariableName
```

The C preprocessor (cpp) is used to first put the file into the correct syntax based on the definitions in *asm.h* before running the assembler. Another change to note is for *immediate* values. Use `IMM (value)` instead of `#value`.

Look at *crt0.S*. The `.text` and `.data` statements tell the assembler that whatever follows is to be loaded in the instruction and data regions of the program, respectively. The `.global` statements declare global symbols that may not be defined in this module. Note that after the first `.text` statement, there is a label called `SYM (start)`, which will be the first instruction in the program. This will be where your program starts, otherwise known as the entry point.

The specification of "`-Ttext 20000`" in the LDFLAGs of the makefile, tells the linker that the first instruction in the instruction segment, or text segment, is to be placed at address $20000. 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? Where does the stack start?

The next major part of the code is commented as `zerobss`. There are two external symbols declared, `_end` and `__bss_start`. They are symbols generated by the compiler. All locations between these symbols are set to 0. 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 `__bss_start` 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.

Once the initialization is complete, the *main* routine of your program is called, which actually starts the program that you wrote. When your program ends or exits, it needs to go somewhere, preferably the monitor. In *crt0.S*, after *main* returns, the *exit* procedure is called. This eventually leads to a `trap #15` instruction, which is what you would normally do in a simple assembly language program to get back to the Ultragizmo monitor. Can you find out where this is done? Look in the directory

```
/local/src/gizmo/lib/libgizmo2
```

You may find the Unix *grep* command useful.

Now compile your program:

**ugsparc1%** `make hello`

and test it out by downloading the resulting S-record file to your board. Section 6.4 (page 79) describes this procedure.

## 3.1 Preparation and Exercise

1. Answer all of the questions above.
2. Test the example *hello* program that you compiled.
3. 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.
4. Test your program in the lab.



# 4 Using nm and the disassembler

In this section, we will use a program called *nm* to look at the symbol table in the object file and *objdump* to disassemble the object file back to assembly language. You can get more details about these commands with the Unix *man* command. Our versions are slightly different, but the Unix man pages for *nm* and *objdump* will suffice.

The versions that we want to run are called `m68k-coff-nm` and `m68k-coff-objdump`, not the normal system version. To make your life easier, you might want to set up the following shell command aliases:

**ugsparc1%** `alias nm68 "m68k-coff-nm --numeric-sort"`
**ugsparc1%** `alias dis68 "m68k-coff-objdump --disassemble"`

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 hello | more`

The fully linked program is in the file *hello*. The flag, "`--numeric-sort`", 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 "d") or the uninitialized segment ("B" or "b"). Look for the symbols `etext`, `edata`, `__bss_start`, and `end`. What addresses are allocated to the variables declared in the program? 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 space is only created on the stack for it when the procedure is invoked. 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*.

Also try `objdump`.

**ugsparc1%** `dis68 hello | more`

Note that sometimes the disassembler gets confused by some non-instructions and it gets out of sync. For example, in this case it does not show the start of *main* correctly. You may need to use the `--start-address=0xHHHHHHHH` flag to specify a hex address to start the disassembler. Try to get it to start disassembling at the start of *main*.

# 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 vectored interrupts available on the DUART.

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 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, page 93 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 *ttydev* assigned to the address of the Ultragizmo DUART. 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. 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 *cint.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 level to zero (0). You can find this procedure defined in *cint.S*. 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. After typing "q" to end the program, fix the appropriate memory location so that your program will run without reloading it.

# 7 Gizmo Communication

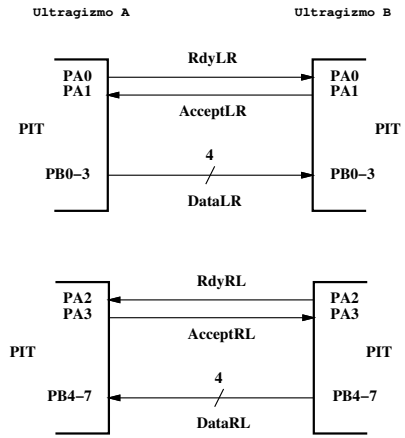
In this exercise, most of your programming should be done in C.

By using the PITs, set up a communication link between two gizmo boards. To make your life easier, do not use interrupts. 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. Take care when setting bits in the ports as you may be changing more than the bit you intend to change unless you are careful. Use the connection scheme shown in the figure below. This will allow you to connect to other boards running different implementations of the program.

   **One other thing to watch.** You will be provided with a long ribbon cable to connect between two different boards. You can directly plug the two boards together using the PIT connectors but be aware that *Ultragizmo A* will use different pins than *Ultragizmo B* for the various functions. You must take this into account when connecting to another board. A well-written program will allow you to switch configurations with little difficulty.




A transmits on PITB 0–3 and receives on PITB 4–7

B transmits on PITB 4–7 and receives on PITB 0–3

**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. To test this out by hand, go through the exercise in Section 8.3.3 (page 105).

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*?