Development Environment

The system described is a **cross-platform development system** with the C cross-compiler, linking with other modules and/or libraries. The executable image is downloaded to the target machine for execution using *FLAMES*.

0.1 Getting Started

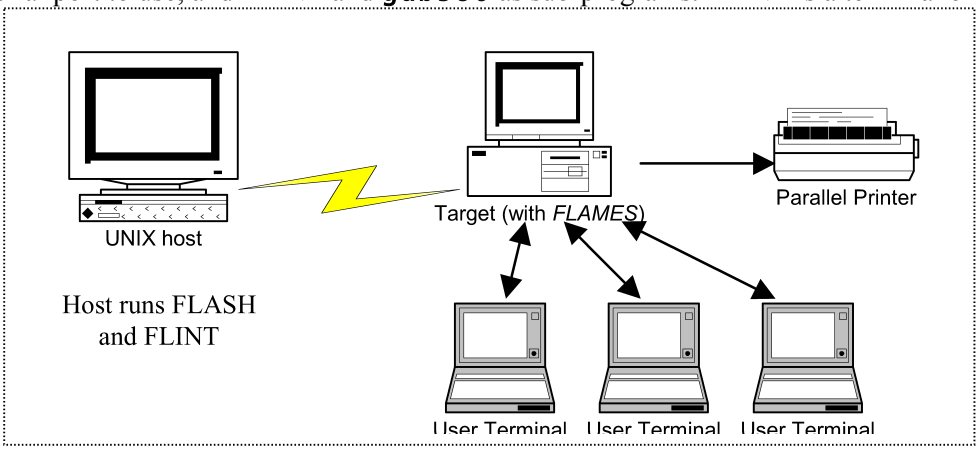
The next two sections provide a short introduction to program development. First is a small sample program for you to type in and run. It will output to the console's display (on the target) using `cons_printf()` and the host computer's display using `printf()`. The **`make`** program is central to good development. It is explained in next section.

0.1.1 Host Tools

The software suite consists of an ANSI C and C++ compiler, assembler, linker, disassembler, *Makefile* maintenance script, boot file builder, and various archiving utilities. Once compilation and linking are successful, SPEDE-X86 commands are used to download the executable module to the target machine, and the ***FLAMES*** monitor is used run the code. Currently the GCC^{x86} system is set up for compiling on a HP-UX host system targeting an Intel x86 CPU. Consideration of the byte order is compensated for by the GCC^{x86} system.

GCC^{x86} provides an ANSI C/C++ compliant environment. However, when you're developing your own operating system, some include files have no meaning without a host operating system. A few GCC-specific extensions are recommended since they simplify interfacing with assembly language. The installed GCC^{x86} software is GNU GCC version 2.7.2.3 and **BinUtils** 2.7 for a HP-UX host, and a target of `i386-unknown-gnu`. The Free Software Foundation, Inc. prevents providers of GNU software from restricting users' rights to its source code.

Custom tools for the host includes **`spede-mkmf`** and the FLASH command shell. The first is a shell script that builds a *Makefile* using the correct compiler with options for cross compiling. A single directory is used to hold one project, where all the source files are compiled into a single OS image. The FLASH command shell manages communication with the target computer. It knows which serial port to use, and FLINT and **`gdb386`** as sub-programs. FLINT is a terminal emulator



for talking to the target. The debugger has been compiled to debug Pentium code, where the host typically is not a Pentium-based computer.

0.1.2 I/O Library Functions

SPEDE provides a small collection of basic input/output routines. These functions are linked with your program, and thus can be called from inside kernel threads and the kernel. All the library routines are thread-safe. For instance, all the output routines save the processor interrupt flag and then disable interrupts upon entry. The interrupt flag is restored upon exit, which normally enables interrupt processing. It is possible that a thread could be outputting a message, and then halfway through an interrupt occurs and some code inside the microkernel prints its own message. Without this protection, the kernel's message would appear in the middle of the thread's message. Restoring the interrupt flag, rather than always enabling interrupts when done, prevents interrupts from possibly being enabled inside the microkernel (which is A Bad Thing™).

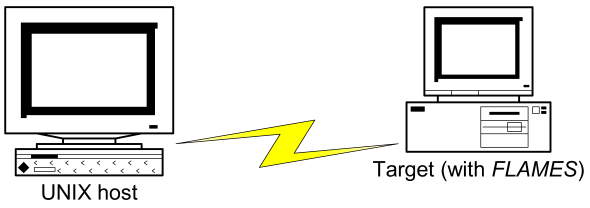
All of the I/O is performed with routines linked into the download image. The *FLAMES* monitor puts the CPU into 32-bit protected mode. There are no calls into the ROM BIOS for I/O, e.g. keyboard, display, or the disk drivers. The BIOS operates in 16-bit real-mode, which is incompatible with protected mode. However, many open source operating systems do have a feature to call real-mode code. This is often used for fallback drivers, such as VESA (located on many video adapters). It's helpful when a device hasn't been developed yet for the new operating system, but a real-mode driver already exists.

Here is a list of some of the input routines. Routines with a "`cons_`" prefix deal with the console display of the target and keyboard. Note, there is no `scanf()` input function. The normal `printf()` and `getchar()` routines use the serial interface to communicate with the attached host computer.

```
/** Returns non-zero if a key has been pressed; returns zero otherwise.
 *    For each function key, a unique callback is invoked. Ie, _kbd_f3()
 *    when [F3] is hit. See <spede/machine/keyboard.h> for prototypes.
 */
extern int cons_kbhit(void);

/** Waits for a keystroke and then returns its ASCII code. If not ASCII, then
 *    the routine keeps waiting. Handles caps-lock and number-lock.
 */
extern int cons_getchar(void);
```

All of these routines are declared in the header file <*spede/flames.h*> so the sample program above includes it. Here are the console output routines.

```
/** Display a single char on the video display. Chars '\n', '\r', '\t', '\b'
 *    and '\f' perform an action. All other chars render themselves.
 *    The backspace action is "non-destructive"; it just moves the cursor.
 */
extern void cons_putchar(int ch);

/** Write a string to the display using cons_putchar() for each character.
 */
extern void cons_putstr(const char * str);

/** Write a formatted string to the console using cons_putchar() for each
 *    character.  Supports standard format specs (e.g., %d, %c, etc.) plus
 *    nonstandard %t (base two) and %b (bitmask).
 *    Returns the number of chars output.
 */
extern int cons_printf(const char * fmt, ...) ;
```



`printf()` output. `cons_printf()` output.

Figure 0-1: `printf()` and `cons_printf()` Destinations

0.1.3 Compiling and Executing a Program

The following sample program will be used to help explain some important aspects of compiling, downloading, and executing a GCC^{x86} program under SPEDE-X86. This program outputs a message to the Unix workstation attached to the "target" PC, and on the console video screen of the target. The source program must be contained in a file named with a ".c" extension – for example, *sample.c*. These extensions will be recognized as a C++ file: "`.cxx`" "`.C`" "`.cc`" and "`.cpp`"

First, you need to modify your UNIX command shell to reference the SPEDE-2000 host tools. This is done by modifying the PATH variable. The easiest way is to add commands to the startup script. For **`csh`**, this is the *.cshrc* file in your home directory.

> ***Note**: Be sure to include both* ~spede/Target-i386/hppa/tools/bin *and* ~spede/Target-i386/hppa/gcc/bin *in your shell's command path before attempting to compile and download this program. If you use **`csh`** or **`tcsh`** you can add the following commands to your startup script (e.g., your* .cshrc *file) to hook into SPEDE-X86 automatically.*

```
# Changes for CSH...
# Next line just for vi users:
setenv EXINIT "set showmode sw=4"

#  Specify Intel x86 as the target CPU
set spede_target=Target-i386
#  Point to host-specific architecture directory for specified target CPU
#  (Below uname is surrounded by back-ticks [accent-graevs])
set spede_arch=~spede/$spede_target/`uname -m`

# If target is available for this host, set access to facilities
if ( -d $spede_arch/tools/bin ) then
      set path=($path $spede_arch/tools/bin $spede_arch/gcc/bin)
      if ( "${?MANPATH}" == 0 ) then
            setenv MANPATH $spede_arch/man
      else
            setenv MANPATH $spede_arch/man:${MANPATH}
      endif
endif
```

Then, the next time you log in, your *.cshrc* will check if SPEDE-X86 is available for the architecture of the target CPU you're interested in on the host you are logged in. If it is, then you can run SPEDE-X86, read its "man" pages, and browse its GNU-style Info pages. The nested "if" statement concerning `MANPATH` checks whether this shell variable is already set or not.

It is recommended that you actually enter the code for the program and go through the steps outlined below. The pound signs (#) should be in the first column of the file. The SPEDE-X86 include files shown in the sample program are stored under the "*spede/*" include sub-directory to emphasize the fact that these include files are not standard C include files (i.e., for the host system).

```
/*  sample.c - Sample Main program for SPEDE      Aug 2001 */

#include <spede/stdio.h>              /* For printf() */
#include <spede/flames.h>             /* For cons_printf() */

int main(void)
{
      long  i;

      i = 111 ;
      printf( "%d Hello world %d \nAAA", i, 2 * i );
      cons_printf( "333 Hello world 444\nBBB" );
      return 0;
}   /* end main() */
```

Suppose you have entered the above program source code into the file *sample.c* and now wish to compile and link the program for downloading to the target machine. The actual commands used to directly invoke the GCC^{x86} compiler and linker are lengthy and are not normally used directly (see the lab manual for details). Rather, you should use a *Makefile* to accomplish the compile and link steps. A *Makefile* is a file containing information ("targets" and "rules") which tells a special program, named **make**, how to compile and link your program. (See the UNIX Tools reference material, or one of the UNIX texts listed in the references, for a discussion of **`make`** and *Makefiles*.) Using a *Makefile* means you always use the correct options, and others can also build your OS using your *Makefile*.

Manual creation of a *Makefile* can be a tedious operation, since the rules for input to the **make** program are somewhat involved. A helper program named "**`spede-mkmf`**" ("**Spede**'s **m**a**k**e

make**f**ile" utility) is available to automatically create and maintain a *Makefile* for your operating system. The utility **spede-mkmf** searches the current directory for assembler, "C" and C++ source files. It determines source dependencies among the files it finds (including automatically recursively following dependencies implied by `#include` statements in the files). **spede-mkmf** is a script in the *tools/* directory. To invoke it, at the shell prompt just type:

```
% spede-mkmf -q
```

(The percent-sign represents the UNIX shell prompt; what the user types is underlined.) As it finds source files in the current directory, it creates "dependency rules" to store into the *Makefile*. Once it's done, just type **make** to invoke the compiler and linker. After some system activity and various GCC^{x86} command executions, the shell prompt should return.

Regarding the sample program, it should be noted that `printf()` sends its output to the UNIX (host) workstation; there is no way to redirect its output since there is no operating system running in the target machine. However, if the terminal window on the host supports it, you can copy the output to the clipboard. Routines with a "`cons_`" prefix send their output to the video display (target machine's video display). The format routines `sprintf()`, `snprintf()`, `cons_printf()` and `printf()` all share the same implementation, and include the non-standard `"%t"` (base **t**wo) conversion for binary values.

Once you have entered a source code program file (e.g., *sample.c*) and then run **spede-mkmf** and **make**. Your directory should include the files *sample.o* and *sample.dli*, which represent the object (binary) file produced by the compiler for input to the linker, and the program image (in **ELF** binary format)[1].

If there were syntax errors, use an editor to correct them, then invoke "**make**" again. For instance if all warnings are turned on for the compiler, it might complain that you're passing a "`long`" type to an "`int`" `printf` format (`%d`). You can correct this by changing the type of "`i`" on line 7 from "`long`" to "`int`".

Once your program is compiled and linked, it can be downloaded into the target machine and then executed. To do this your workstation must be directly connected to a target computer (that is, downloads cannot be accomplished from a remote location on the network).

In order to download the program to the target machine you must first start the monitor/downloader program running on the target. The monitor/loader program is stored on the hard disk of the target computer. Boot the target machine to the monitor/loader. Its *autoexec.bat* startup file will invoke **FLAMES.BAT** for you. *FLAMES* uses COM1 to communicate with the host computer. (Note, because the target monitor program isn't stored in ROM it can be corrupted very easily. If the target stops responding, it may be necessary to reboot it.)

To download your program from the host computer, you must invoke **FLASH**, the **Fl**ames **A**ccess **Sh**ell on the host computer. FLASH allows your workstation to communicate with the monitor program of the target machine; the monitor in turn allows you to execute and control a downloaded program. To start FLASH, type the command:

```
% flash
```

[1] The download file name is "*sample.dli*" after the subdirectory the source files are stored in. The name of the downloadable file can be easily changed by editing the variable `OSNAME` in the *Makefile*. If you change this variable, there must not be any trailing spaces after your OS name.

at the UNIX host workstation (HP/UX workstation, not the target machine). You must be logged into the workstation. If there is an error, you might be logged into another computer. When FLASH displays its prompt, type the command

FLASH> download <*filename*>.dli

where <filename> is, for example, *Sample.dli*. FLASH should display the size of the file in blocks, and then display each block number as it is downloaded. When the entire program is downloaded, the FLASH prompt returns. Once you give a file name to the download command, *FLASH* will remember it for the next download or debug command. If you give the DLI file on the UNIX command line, then it becomes the default for all download and debug commands.

The system also provides an Ethernet download capability. Normally *FLASH* uses the serial line for downloading. (For smaller programs, the startup time might make serial download faster.) However, if you set the shell environment variable "SPEDE_ETHER" to "YES" then *FLASH* will probe for an Ethernet hookup first. (It uses the serial link to do this.) It will fall back to a serial download if the Ethernet cable is not hooked up to the target, or the target cannot RARP for its IP address. To have this as your default, add the following line to your *.cshrc* file:

setenv SPEDE_ETHER YES

In order to execute the downloaded program, it is necessary to connect directly to the target machine (as opposed to being connected to the command shell of FLASH). The command FLINT will run a terminal emulator on the host that uses the serial link. This is done by typing the command:

FLASH> flint

at the FLASH prompt. Once you get the `FLAMES>` prompt (generated by the monitor running on the target machine), you can enter commands at the UNIX workstation. They will be sent to the monitor on the target machine for execution. (To see a complete list of the monitor commands, enter "?" at the `FLAMES>` prompt. See the chapter on *FLAMES* in the lab manual for details.)

Once you have downloaded your program and are in FLINT, you can run the program by typing the command "`g`" (for "go using the default starting address"). You should verify that the sample program executes correctly; there should be one message on the workstation window screen and another on the console screen of the target machine. To exit from FLINT back to *FLASH* , type control-X, followed by *ENTER*, or just control-C by itself. To exit from *FLASH* back to the host OS, type "`quit`".

After you download your OS, you can either run it directly using FLINT or with the debugger **`gdb386`**. When your program starts running, it uses it global variables, e.g. updating their values. Once it is done running, the global variables are, in a sense, "used up." If you want to run your program again, you must download the image again. If you tried to run it again, your global variables would have incorrect values, e.g., values are not what you program expects it to be initially. This applies also to "`static`" variables, local to a function.

0.2 More on Makefiles

In this textbook, you will notice the filenames *Makefile* and *makefile* used interchangeably[2]. The **`make`** program looks for both forms. If you don't like either name, the **`-f`** option lets you directly specify the make file name.

[2] With UNIX, letter-case in a filename is significant, and the two forms can coexist in the same directory.

0.2.1 Dependencies

As mentioned above, the commands to invoke GNU GCC directly are a little cumbersome. This is one of the advantages of using **make**. In addition, as programs grow larger and more complex, it becomes difficult to keep track of which modules depend on which files; i.e., which modules need to be recompiled when a header file is changed. Fortunately, **make** can be used to keep track of such relations. This is accomplished by adding information to the *Makefile* describing the dependencies between the various source files. Once the *Makefile* contains this information, when **make** is invoked it will automatically recompile those modules (and only those modules) which need updating prior to linking the program.

Creating the dependency information for inclusion in the *Makefile* can be tedious, especially if you are unfamiliar with the format of a *Makefile*. To help out, there are rules in the *Makefile* that can automatically generate *and update* its dependency data. Typing the command

```
% make depend
```

will update the dependency relationships among all source files *in the current directory*. It also updates the source file list in the *Makefile*. The dependency information in the *Makefile* must be updated each time the dependency structure of the program changes—for example, whenever a new source or header file is added.

0.2.2 Structure of the Makefile

A *Makefile* contains variable assignments, comments, and rules. Most often, the *Makefile* tells **make** what to do. For many applications, it tells **make** how to compile and link a program. For SPEDE, it also has rules for disassembly and downloading the image to the target (this last rule is not used often). Comments begin with the pound sign (#) (also pronounced "sharp" or "hash mark" or just "hash," e.g., C's "hash define" directive).

Each rule has three parts: the target name followed by a colon (:), an optional list of dependencies (prerequisites), both on the same line and then a series of *TAB* indented lines containing commands to execute. The dependencies can be a mix of files and other targets. A blank line or line without a leading *TAB* terminates the list of rules. A typical *Makefile* has the following appearance (line numbers are to the left). By default, it builds the executable "*sample*" from "*sample.c*" which depends on the "*defines.h*" header file.

```
1.          # This is a sample Makefile
2.          CC = gcc -Wall
3.          .PHONY :    all help
4.
5.          all : sample
6.          help :
7.                echo "'make all' to rebuild"
8.          sample :    sample.c defines.h
9.                $(CC) $(CFLAGS) -o $@ $<
```

*Note: You **must** use a TAB indent on the command lines in a* Makefile*. Attempting to use spaces instead will lead to confusion, syntax errors, and assorted personal maladies. When a* Makefile *is printed, it's impossible to tell the difference between spaces and tabs.*

Well, this *Makefile* is a little fancy. Line 1 is a comment. Line 2 sets the variable "CC" to use the GNU C compiler to compile with all warnings displayed. Line 3 tells **make** that the targets

"all" and "help" doesn't really specify a real file, they are just "naming handles" for the rules. (Leaving it out means if there is a file named *all* in the directory, it will be considered when testing to see if *all* is up-to-date relative to *sample*.)

Line 5 is the first rule, named "all" (the name "all" is just a convention). Since it is the first rule, it is the default when **make** is invoked without an explicit target name. It has the single dependency "sample" which is a file name (compiled executable), and there are no commands in this rule. However, since "all" depends on "sample", make will do its commands first before finishing with the "all" rule. The second rule causes **make** to echo a short help message in response to the command "`make help`" from the user. Here the command is just **echo**.

The last rule is on lines 8 and 9, and will recompile the sample program if needed. It will check the file modification times of *sample*, *sample.c* and *defines.h* to see if the last two have been modified after the first file. If yes, then the executable is out-of-date relative to its source code. Therefore, make will execute the commands to update the target file. This means invoking GCC to compile and link *sample*. Lines 7 and 9 are indented with a single *TAB* character.

Line 9 is the single command line for the "sample" rule. It references four different variables inside of make. The first two variables are named, and are referenced by surrounding them with parenthesis and prepending a dollar sign ($). The last two automatic variables are single characters. The at-sign (@) is substituted with the rule name, in this case "sample" which happens to be the output name we need. The last variable is the less-than symbol (<) and stands for the first dependency name, in this case it's the source code filename. Automatic variables cannot be set; rather **make** sets them using information about the current rule. Because they are single characters, automatic variables don't need parenthesis around them.

0.2.3 Makefile *Rules for Assembly Language Source Files*

The programmer can include assembly language files to be assembled and linked as part of a program. Any assembly source files you write must have names that end in *.S* (capital letter S), for example "*io.S*". These files will be processed by **cpp** before assembly, so they can include C-style comments and `#include` directives.

The *Makefile* generated by **spede-mkmf** contains production rules that will create an Intel x86 disassembly. There are two ways of doing this. The first takes a C source file and re-generates the assembly code for it. These generated assembly files have a *.asm* extension. For example, if you want to inspect the assembly code produced from a single C source file named *sample.c*, you would type:

```
% make sample.asm
```

This will create a file named *sample.asm* containing the assembly code corresponding to *sample.c*. (Again, the percent-sign is the UNIX shell prompt.) A *.asm* file generated in this way corresponds to a *single* C source file. The *.asm* extension means this file isn't a legal assembler file. It contains assembly and C code mixed together; you cannot assemble this file as-is. Files with the extension *.asm* are removed as part of the "`make clean`" command.

The *.asm* file also contains useful "symbol table" (stab) information. For example, each group of assembly instructions corresponding to a single C source code line starts with a line of the form

```
.stabd  68,I,N
```

where "*N*" is the line number in the original C source file. Encoded in the **stab** information is where local variables are stored: either stack frame or in a register.

The second way is to get an assembly listing of the entire executable image. To generate an assembly language file corresponding to a complete compiled and linked program (e.g., the *Sample.dli* file), type:

```
% make text
```

This produces an assembly language file *text.asm* from the *Sample.dli* file (which must already exist). This is extremely useful since it gives you the memory addresses for the assembly language instructions corresponding to C statements. It intermixes symbolic addresses, assembly code, and statements from the source files. Unfortunately, it doesn't give you source file names. Here is the command it executes for you (one very long line):

```
objdump386 --disassemble --file-headers --reloc \
          --source Sample.dli > text.asm
```

Note the use of two dashes before each option. This is the GNU style for their long name options. Most UNIX programs have single letter options preceded by a single dash.

0.2.4 Additional Features of the SPEDE Makefile

Normally, the *Makefile* will cause the compiling of all source files followed by linking of your operating system. Running `make` with no parameters does this. However, if you just want to check the syntax of some file, say, *sample.c*, type:.

```
% make sample.o
```

where *.o* is the object module extension. This compiles *sample.c*, if required, but does **not** link it into the final executable module.

As your program evolves and new header files are added, type "`make depend`" to rebuild the header file dependencies stored in your *Makefile*. For hints, type "`make help`" to see what targets are part of the *Makefile* ("help" is one target in your new *Makefile*). On UNIX systems, you must use GNU `make` ("gmake") to compile, not the default (System V) CCS version.

When you're unsure of what you have, or simply want to clean up your directory, the command:

```
% make clean
```

will remove all those files that can be automatically regenerated. In addition, when you're confused about the *Makefile*, type `make help` for a brief listing of capabilities available! Both the targets "clean" and "help" are part of the specific *Makefile* built by `spede-mkmf`.

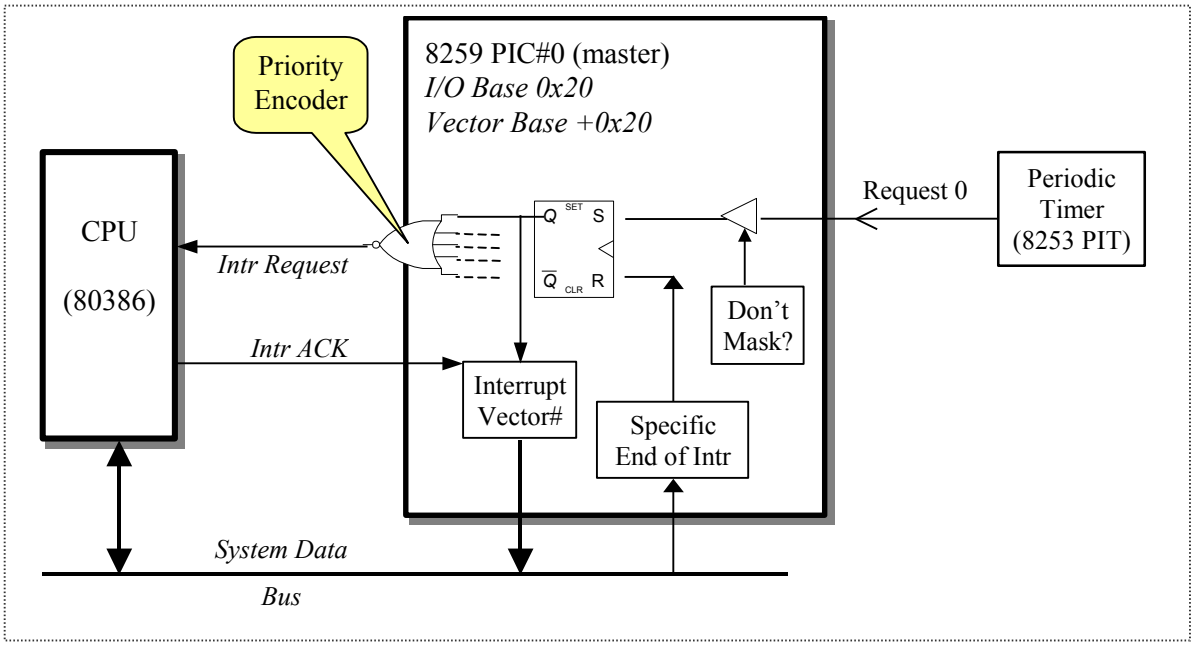
If you want to add arbitrary commands into a *Makefile*, create a file named "*local.mk*" and store them there. After all the rules and variables are defined, but before the default target is defined, this file will be included. It is useful, say, when you need to convert data files into a form the C compiler can parse. Use a double colon ("`::`") after rules when you want to stack them on top of existing rules (targets). When user applications are added to the operating system (invoked from the command shell), a *local.mk* file uses this feature to add commands to the "clean" target.

0.3 Introduction to the PC/AT Interrupt Architecture

Before delving into an actual program that handles the periodic timer interrupt, lets review the interrupt mechanisms in the PC/AT architecture. This system will be covered in more details in the device drivers chapter. What follows here is a description of the logical model of the interrupt processing hardware and associated data structures. It is the "view from 50,000 foot" of things.

The Intel processor has a single flag indicating whether or not it will handle interrupts. This is the `IF` flag within the `Eflags` register. But this is only one input signal. The Intel system has an external multiplexer of various hardware interrupt requests. In the PC/AT system, there are two devices that handle a total of 16 inputs for the CPU. The **Programmable Interrupt Controller** (PIC) holds the request from different devices until the CPU indicates it is done handling the current interrupt request. This is important for giving priority to certain devices. It can also hide a device's request if the CPU just doesn't want to be informed when the interrupt request occurs. Your operating system starts out by paying attention to a single request and ignoring all others.

On the hardware side are the PIC chips that communicate with the CPU; one the software side is the interrupt descriptor table (IDT). This table provides the connection between the hardware interrupt request and code to handle that request. It stores the address of a function to execute when the CPU receives a particular interrupt request. (The example in the next section shows how to setup and service the periodic timer interrupt, IRQ 0.) When the CPU is ready to handle the interrupt, it will push the (instruction) location of the next instruction (offset and segment), along with current "flags" value. It will then jump to the interrupt handler. Described below, the code in "*entry.S*" is the first-level interrupt handler.



0.3.1 Interrupt Support Hardware

Here is a diagram of the programmable interrupt controllers and the CPU. It shows the insides of one request line. The periodic timer is free spinning, and generates many interrupts per second.

Inside each programmable interrupt controller is logic to latch the request inputs. But first, each source can be masked out. This is in addition to the CPU postponing the handling of interrupts. Each 8259 contains arbitration logic so when a higher priority request is being handled, a lower priority one doesn't generate an interrupt request to the CPU. The lower priority interrupt won't occur until the kernel has acknowledged handing the current request.

When the external device issues a request, it can be a single pulse (e.g., the periodic timer issues pulses 100 times a second to the PIC chip). The interrupt controller latches this pulse using a flip/flop. The outputs of all the flip/flops are OR'ed together to create the interrupt request indication. When the CPU is ready to service the request, it responds with an interrupt acknowledge response. At this point, the PIC uses the data bus to reply with the request vector number of the highest priority request. This vector number is the sum of the controller's preprogrammed "vector base" and request line (0 to 7).

0.3.2 Interrupt Support Software

At each instruction boundary, the CPU looks for an external interrupt request indication. The hand-shaking between the interrupt controller and the CPU is handled by logic within the CPU. Program logic can disable this check by executing a **`DI`** opcode (disable interrupts), and the CPU will no longer generate the acknowledge response. Handling begins after the interrupt acknowledge, wherein the interrupt controller sends out an interrupt vector number. This number will be used to index directly into the interrupt descriptor table (IDT). Once the CPU receives the request vector, it starts interrupt processing. It begins by pushing the **`CS`**, **`EIP`** and **`EFlags`** registers. The CPU then reads the base address of the interrupt descriptor table, and uses the vector number as the index to find the appropriate descriptor. Since each descriptor is eight bytes, the vector number is multiplied by eight and then added to the IDT base address.

The CPU reads the interrupt descriptor entry. From it, the CPU extracts the segment and offset of the interrupt handler. The code in the next section uses `get_cs()` for the segment, and the address of the function for the offset. The function `set_exception_handler()` builds up the entry in the IDT.

External interrupt request #0 maps to interrupt vector 0x20; the PC ROM BIOS maps it differently. Interrupt vectors starting at 0x30 will be used for kernel services.

0.4 Handling the Timer Interrupt

This section describes code to handle timer interrupts on a Pentium computer. The program has two components: a foreground "thread" constantly increments a memory location in the text video display, while the timer interrupt service routine counts seconds, outputting a character once a second. The *FLAMES* monitor sets up the timer, however, since interrupts are disabled, it doesn't reach the application until it's ready. The program demonstrates the following:

- hooking the interrupt vector,
- calling assembly code to save all registers (the context),
- setting up the "C" environment in assembly and then calling it to do some useful work,
- returning to whatever the CPU was executing before the timer interrupt.

It is a stand-alone program, *not* an example of how an OS handles timer interrupts and thread dispatching. However, the program does contain many of the interrupt handling elements needed by an OS to do these things. One exception is the `EI()` call, which enables the processor's

handling of external interrupts. For the OS, this call must be removed. Interrupts will be enabled when each thread is dispatched by the IRET instruction. Interrupts are always off inside the microkernel.

The interrupt handling in this small example is what programmers think of when writing a normal interrupt service routine. To compare this with an operating system, all the code from `timer_entry` to the **`IRET`** in the ISR will be "inside the kernel" of the operating system.

This example consists of three files: *main.c* defines the basic application that executes in the "foreground," *service.c* has the "C" code to handle the timer interrupt, and *entry.S* contains assembly code to save registers and call the timer interrupt service routine.

The first part of *main.c* includes the proper SPEDE header files, defines a new function pointer type useful for pointing to interrupt service routines, and declares the first-level interrupt handler as an external function with "C" style linkage. If compiled as C++ code, it won't name-mangle this external function[1]. This is taken care of by the `__BEGIN_DECLS` and `__END_DECLS` bracketing (each starts with two underscores and are defined in <*spede/sys/cdefs.h*>). Assembly code always uses "C" style linkage. The file also defines one global variable, a pointer to the Interrupt Descriptor Table (IDT) the CPU is currently using.

```
/*  main.c - Hook timer intr, exit when user hits key    July 2002 */
/*   $Id$   */

#include <spede/flames.h>             /* For PTR2INT() macro */
#include <spede/machine/io.h>         /* For outportb() */
#include <spede/machine/proc_reg.h>   /* For get_cs(), get_idt_base() */
#include <spede/machine/seg.h>     /* For i386_gate, fill_gate(), pseudo_desc */
#include <spede/machine/pic.h>        /* For IRQ_VECTOR() macro */

/* Ptr to function take no parameters, returning void, nothing. An ISR */
typedef  void (*PFV)( void );

__BEGIN_DECLS
/* first-level handler entry (assembly) */
extern void        timer_entry(void);
__END_DECLS

/* Ptr to start of Interrupt Descriptor Table (IDT). It's really an array
 * of 256 "i386_gate" structures.
 */
struct i386_gate   idt_table[];
```

The interrupt gate created by the following function can only be accessed from ring 0, since the privilege level was not specified (technically, it specifies access from rings <= 0). To allow a ring 3 user-mode program to use the descriptor, use `ACC_INTR_GATE | ACC_PL_U` (access privilege level: user-mode). The function `fill_gate()` is a SPEDE library routine which takes a pointer to memory to store a descriptor, the offset and code segment of the handler, and a code to state what kind of descriptor to store.

[1] To see the name mangling, comment out these lines and recompile. The linker will alert you to an unresolved reference. Sometimes the linker reverses the name mangling to display the function prototype (the GNU linker does this). Use **`nm386`** to inspect the object file for the unresolved symbol's true name.

```
/**
 *     Builds an i386 interrupt gate containing the specified "handler"
 *     (ISR entry) address.  It stores that gate in the Interrupt
 *     Descriptor Table entry indexed for a specific exception/interrupt.
 */
void set_exception_handler(int exception, PFV handler)
{
       /*  Get address of this particular interrupt descriptor entry: */
       struct i386_gate * gateptr = & idt_table[ exception ];

       /*  Build a valid Interrupt Gate in this IDT entry: */
       fill_gate( gateptr, PTR2INT(handler), get_cs(), ACC_INTR_GATE, 0 );
}   /* end set_exception_handler() */
```

The last part of *main.c* is the `main()` function, that first sets up the timer interrupt hook through the interrupt gate. It uses a SPEDE function to read the base address of the Interrupt Descriptor Table (IDT) from the **`IDTR`** register and stores it into the `idt_table` global variable. Once this variable is set, the program can hook the timer interrupt vector. The IDT is indexed by an interrupt number. Each entry in the IDT describes the actions the CPU must take. For the periodic timer interrupt (IRQ 0), we want an interrupt gate to call our assembly code (first-level interrupt handler). An interrupt gate will clear the interrupt flag, preventing further interrupts.

"Hooking an interrupt" requires discovering where the IDT is located in memory. Each entry directs the CPU to a service routine for a particular interrupt number. The system default traps into ***`FLAMES`*** where it prints a diagnostic message. To hook the interrupt means pointing to a service routine in your code. This is the job of `get_idt_base()` and the aforementioned `set_exception_handler()`. The latter is used for hooking both hardware and software interrupt.

For hardware interrupts, additional setup is required. After installing a function pointer into the IDT entry, the interrupt controller must be told to enable the hardware interrupt. The programmable interrupt controller (PIC) allows the timer interrupt to reach the CPU by unmasking only IRQ 0. The PC/AT architecture has two of these 8259 chips, and the code below treats the mask as a single 16-bit value. It is done for completeness. Since no interrupt request signals are unmasked in the slave PIC (at I/O base 0x00A0), the second `outportb()` could be removed.

Next, the code enables interrupt handling in the CPU, and then loops until a key is pressed. The "while" loop also increments a cell of the screen memory. This is feedback to the user that the program is still alive. The key press is pending, so `cons_getchar()` must be called. When `main()` returns, the SPEDE runtime code will disable interrupts and clean up.

```
/**
 *   Get address of IDT then hook the periodic timer IRQ vector.  Unmask
 *   interrupts.  The timer ISR will display ticks while we wait for a
 *   keypress to exit.  Increment screen memory while waiting.
 */
int main()
{
      volatile uint16 * vidmem = (uint16 *)0x0B8000 + 80*20;   /* Text line 20 */

      /* First, drain any stray keypresses: */
      while( cons_kbhit() ) { (void) cons_getchar(); }

      /* Find out where FLAMES placed the IDT array: */
      idt_table = get_idt_base();

      /*  Hook the timer interrupt. Set the timer vector (IRQ 0)
       *  to point to the assembly entry point for the timer.
       *  Now have pointer to existing IDT, fill in timer slot.
       *  The macro IRQ_VECTOR() converts from IRQ# to vector#.
       */
      set_exception_handler( IRQ_VECTOR(IRQ_TIMER), timer_entry );

      /*  Unmask IRQ 0 (the timer interrupt) while keeping all other IRQs
       *  masked (ie, disabled) on the master i8259 PIC (interrupt control
       *  unit zero).  Note, a 0 bit = enable.
       */
      outportb( ICU0_IOBASE+1, ~ 0x01 );     /* ~0x01 = 1111_1110 */

      /*  Mask out all IRQs on the slave i8259 (one). */
      outportb( ICU1_IOBASE+1, ~ 0x00 );     /* ~0x00 = 1111_1111 */

      /*  We're ready to enable interrupts and start handling the timer.
       *  Loop waiting for a keypress on the console, getting interrupted
       *  occasionally.
       */
      EI();                      /* <-- REMOVE FOR OS CODE! */
      while( 0 == cons_kbhit()) {
            IO_DELAY();
            *vidmem += 1;
      }   /* while no key pressed.. */
      (void) cons_getchar();          /* Eat the keypress. */

      /*  When we exit, FLAMES will disable interrupts for us. */
      return 0;
}   /* end main() */
```

Once the interrupt controllers are setup, we enable handling of interrupts inside the CPU. Before this, if the PIC said there was an interrupt request the CPU would ignore it. The function `cons_kbhit()` just checks if any key has been pressed; it does not block waiting for a key press. The while loop causes a text character cell on the display to continually increment. This provides a very low overhead feedback mechanism indicating that the program is running. Once a key is pressed, the program exits, after which the ***FLAMES*** runtime will disable interrupts and restore things.

As a way to test this program, you can leave out the `EI()` instruction. Without it, the timer interrupt will never be acknowledged by the CPU. However, the screen should still flicker, and hitting a key on the target keyboard should exit the program. (The target console routines use polled-mode input.)

The second source file defines the high-level timer interrupt service routine. This code is called by the first-level interrupt handler (FLIH) entry assembly code, thus the use of

__BEGIN_DECLS and __END_DECLS again (these symbols begin with two underscores). The C code does whatever processing is required (in this case counting "ticks") and then dismisses the clock interrupt.

```
/*  service.c - Handle timer interrupts                    July 2002 */
/*   $Id$   */

#include <spede/flames.h>             /* For cons_putchar() */
#include <spede/machine/io.h>         /* For outportb() */
#include <spede/machine/pic.h>        /* For 8259 PIC defines */
#include <spede/time.h>               /* For clock_t, CLK_TCK */

clock_t      tick_count = 0;    /* Count timer ticks, whatever the rate */

__BEGIN_DECLS
void timer_ISR(void);           /* Declare C linkage since called from assem */
__END_DECLS

void timer_ISR()
{
      /*  Output a character once every second. */
      if( 0 == (++tick_count % CLK_TCK) ) {
            cons_putchar('X');
      }

      /*  Dismiss the timer interrupt. Send a "specific End-Of-Interrupt
       *  for IRQ 0" command to the Master Interrupt Control Unit.
       */
      outportb( ICU0_IOBASE, SPECIFIC_EOI(IRQ_TIMER) );
}   /* end timer_ISR() */
```

The last statement of timer_ISR() will dismiss the interrupt indication from the interrupt controller (8259). The periodic timer is setup by ***FLAMES*** to provide a steady stream of interrupts; there is no enabling that needs to occur. When an interrupt request first goes to the interrupt controller, it sets a flip/flop. This allows the device request to be just a pulse. When the CPU decides to handle the interrupt request, the controller presents the request number to the CPU, added it to a stored "base interrupt vector" value. For your first operating system, IRQ *L* maps to interrupt 0x20+*L* (this is not the usual IBM ROM BIOS setting). There is no count of how many times the device made a interrupt request, only that it has made at least one request.

When the software interrupt handler has finished, it must tell the interrupt controller it has handled the request. This is done by sending a "dismiss indication" for the interrupt when the handler servicing. In the code above, IRQ 0 is dismissed. This resets the request flip/flop in the interrupt controller. Now, if subsequent requests occur, there will be another interrupt request to the CPU.

Lastly is *entry.S*, which contains the assembly code called via the interrupt gate. The CPU executes this first after acknowledging the interrupt. The ENTRY() macro does the bookkeeping necessary to define a function in the code segment. After saving all the general registers using the **PUSHA** instruction, it prepares for the "C" code environment. The only detail for the Intel X86 is clearing the direction flag (do not confuse clear direction, **cld**, with disable interrupt handling, **cli**[2]). Once this is done, a **CALL** to the "C" code occurs.

[2] Again, a better name is "postpone external interrupt handling." Faults will still be handled by proper interrupts.

The macro `CNAME()` is used to reference an external function (with C language linkage) defined by the C compiler. Some systems prepend an underscore to C function names. However, with ELF object files this is not done. The macro helps to declare a scope for the label, in this case it's "global."

```
/*  entry.S - Handle timer interrupts              July 2002 */
/*   $Id$   */

#include <spede/machine/asmacros.h>   /* For ENTRY() and CNAME() macros */

ENTRY(timer_entry)
            pusha                     /* Save all general regs */
            cld                       /* Prepare "C" environment */
            call  CNAME(timer_ISR)
            popa                      /* Restore all general regs */
            iret
```

The direction and interrupt flags are stored in the flags register. The CPU will push **`CS`**, **`EIP`** and **`EFlags`** when it handles the interrupt, and then clear the interrupt flag. If there is a code ring change, it will also push **`SS`** and **`ESP`**, then read new values for these registers from the task state segment (TSS). Then it executes the assembly code in the FLIH from *entry.S* (above). The **`IRET`** will restore these three registers. This code assumes all code shares the same segment selectors. If this is not the case, you must also save the segment selector registers, **`CS`**, **`DS`**, **`ES`** and **`SS`**, then set them to values used by the kernel. Upon return to a user thread, the previous segment values must be restored along with the general register values (this will be done in the kernel's entry code). If the popped **`CS`** has a ring value (CPL value in lower two bits) different than the kernel ring, the CPU will also pop **`ESP`** and **`SS`**.

Note that in the assembly code there are no external function declarations. The assembler just assumes undefined symbols are external (e.g., "`timer_ISR`"). If you don't have the named routine (e.g., there is a typo in your source file), the linker will display a diagnostic message, not the assembler.

0.4.1 Running the Timer Example

Once the three files are typed in, you should have them in a directory all by themselves. This will be where you compile and download the image. The first step is to have SPEDE generate a *Makefile* for you by typing:

```
% spede-mkmf -q
```

(The percent-sign (`%`) is the shell prompt, what you type is underlined.) (The "`-q`" option means don't ask to create a *Makefile* if none exists.) This will examine the C and assembly source files, determine file dependencies, and build a *Makefile* for you. Now type "**`make`**" to compile and link your example. When done you will have a bunch of object files in the directory along with a file ending in "*dli*" which is the Pentium executable. If there are compile warnings or errors, please fix them before proceeding.

This DLI file cannot run on the host computer. It must first be transferred to the target and then executed. For this use FLASH. First make sure the host computer you are using is properly hooked to the target computer. At the minimum, a serial connection is required. Make sure ***FLAMES*** is running on the target. For development, you can modify the machine's *autoexec.bat* file (MS-DOS startup script) to always run ***FLAMES***. (Do not attempt to run a fancy OS on the

target when running *FLAMES* there. This will get in the way of the protected-mode code used by *FLAMES*.)

If you run FLASH with a filename parameter, it will take that as the default for downloading to the target.

```
% flash sample.dli
```

This will get you into the FLASH command shell. Next, type "down" to begin the download process. If FLASH cannot communicate with the target, it will display an error message. After that is done, type "`flint`" to connect directly to the monitor running on the target. At this point, the program has been stored into the memory of the target computer. You can dump memory, examine the CPU's registers, or probe the PCI bus. Issuing the start command "g", starts running the program.

Immediately you should see one of the characters on the display changing rapidly. This is the while loop of the program (bottom of the `main()` function). Also appearing, at a rate of one per second, should be upper-case X's. Immediately to the right of the "X" will be the cursor. To exit this program, tap a key on the keyboard of the target. This program listens for a key press to know when to exit. If you don't put an exit mechanism into your program, you must reset the computer. The normal control-alt-delete sequence will not work because the ROM BIOS routine that normally handles keyboard interrupts is not being used.