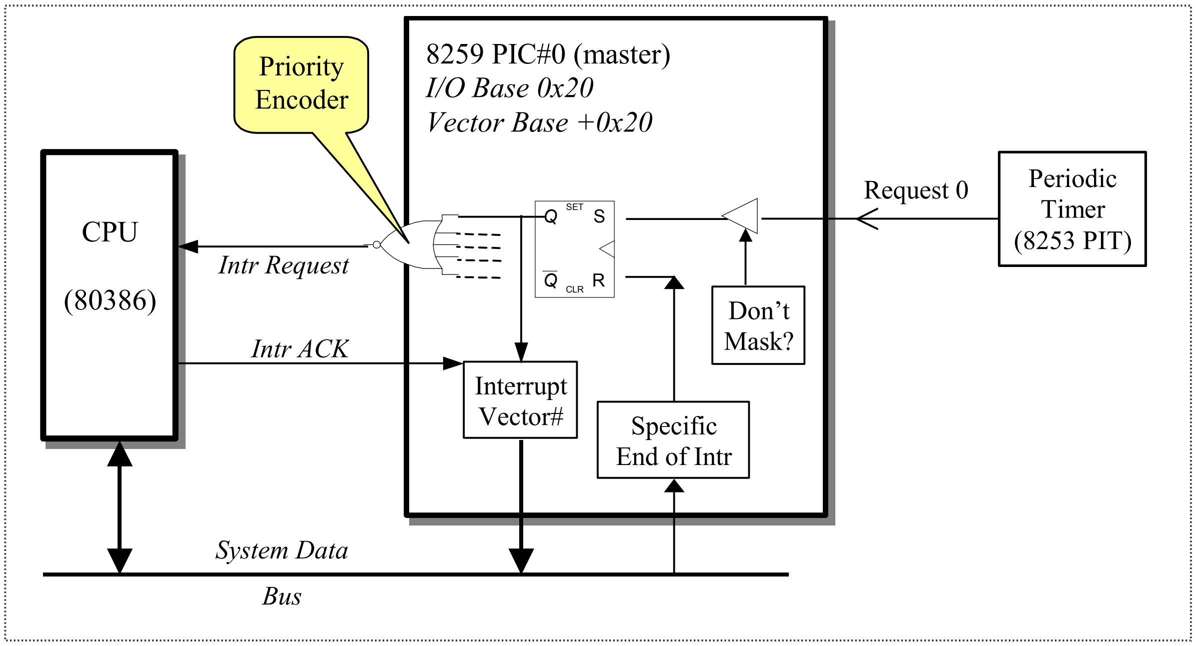
0.3 Introduction to the PC/AT Interrupt Architecture

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~~