CS452/CS652
Real-time Programming

Course Notes

Chapter 1

Lab information

1.1 What to do for various problems

1.1.1 What is wrong?

Problem: Terminals down, UNIX machines down. Use the normal problem reporting mechanisms. If it is something low-priority (i.e., one machine of three is broken), then use the `gripe(1)` command to report the problem, or `tgripe(1)` if it is a hardware problem with a terminal. If it is something high-priority, like all the UNIX machines being down, tell the operator. You can do this during production hours by buzzing at the door to MC3015 near the printer output. Equivalently, you may call x3559 to reach them inside. Outside of regular hours you can call security (x4911) and explain to them that there is a serious computer problem. If you convince them it is serious, they will call the appropriate person at home, waking them from blissful slumber. That person may choose to come in and fix the problem (but they aren't paid to do it, so don't get your hopes up).

Problem: Software configuration. Use `gripe(1)` command to report the problem with as much detail as possible. Do this even if it is high priority, since the operator can't help in most cases. If it is outside of production hours and it's a serious problem (example: close to deadline, no-one can work at all), then you can try looking for a MFCF software person, in the MFCF corridor (beside the lab). Appropriate begging may inspire them to help you, but in all likelihood they already are working on the problem (if it is system-wide), or else can't help you.

Problem: Lab equipment is broken. Report the problem to the hardware design lab. You can do this by sending email to `hardware@math` or `cs452@student.math`. During business hours, you can go into MC3061, and ask politely for help, or call x5471. Naturally, it is your best interest to report the problem in great detail, so that whoever looks at it can figure out what you are complaining about. This means you must give the number and location of a malfunctioning train switch, in addition to how it is failing. Please do not say "the robot seems to be broken today": We expect you to be able to trouble-shoot things to some degree.

Problem: Personal safety threatened. Call the campus police at x4911. Do this if you see suspicious things at three o'clock in the morning. Please do this if you see people carrying equipment away from the other labs. If one of your classmates has a mental or health emergency, x4911 is the correct number to call, not Heath and Safety.

Note: These mail addresses and phone numbers were correct at the time of writing, however they are subject to change.

1.1.2 Summary: Contact addresses

Problem	Email/command	Extension	Physical
Unix software	`gripe(1)`	x6059 (consultant)	MC3011
Lab terminals	`tgripe(1)`		
Unix/network hardware	`gripe(1)`	x3559 (operator)	MC3015
Lab equipment	hardware@math	x5471	MC3061
After hours crisis		x4911 (security)	

1.1.3 Conventions used in this manual

There are a number of conventions that will be used throughout this document.

- On-line manual pages are given with their section numbers following the name: `libsvga(1)`. Use the program `man(1)` to read them.
- All numbers are in decimal unless otherwise noted. Hexadecimal numbers are proceeded with `0x`, just like in 'C'.
- Eight bits is one byte. Sixteen bits (two bytes) is a short-word. Four bytes (32 bits) is a word. Since the *i486* is a 32-bit machine, its word size is 32 bits. Some library functions refer to "quadwords" which are 64 bits long.
- Bits are numbered, starting at zero, from the least significant.

1.1.4 Extra manuals available

These manuals are present in the Train Lab for use by students.

1. WY-75 Display Terminal Quick Reference Guide
2. VT-240 Owners Manual
3. WYSE 99GT User's Guide
4. IFX Indexer Guide (1 copy last page 71/72 loose)

In addition to the above manuals and this lab-book, the Hardware Lab may have copies of the technical manuals for the components of the computer systems you will be using. These manuals are typically of the nature that you'd only want them if you're doing something really funky with the hardware. If that is you intention, they can be borrowed: just ask.

1.2 Lab equipment

1.2.1 General

Please do not power off the train or serial analyzer unless necessary. Warm up problems will affect your program responses. Power cycling shortens the life of most electronic equipment.

1.2.2 Serial analyzer

There is a serial analyzer in the lab. Normally it is connected to monitor the train. It's quite user friendly despite appearances. The only buttons you need to use are the "mode" and "format" buttons (to select the two configuration menus), and the "field select" and direction arrow keys to move around the menus. Remember you want `HEX8` for the train.

The details as follows.

```
FORMAT screen:
      HEX-8
      ASYNC-1
      2400
      EVEN PAR
MODE screen:
      MONITOR
      TX CHAR SEQ
      CONTINUOUS
    TRIG =
      OFF
```

Once these are set up, hitting "run" is the only thing left to do. The analyzer does have a memory, so if you hit "stop", you can use the direction arrows to scan through 2k of back samples.

The serial analyzer is now officially old enough to vote and is suffering the standard problems associated with silicon dementia. If there are problems with your serial connections try turning off or removing the analyzer from the train. Wherever possible, avoid shutting off the analyzer if it is in a working state.

1.2.3 Window and Curtains

The window in the lab was installed so we could show off our pretty systems to the general populous. It is also quite handy so friends/better-halfs can locate you during your long vigils.

Previous terms found solace in a Nerf-Arrow gun, kept loaded and firing at inquisitive watchers. We don't recommend random violence, but it does help sometimes.

When a chair breaks, please just stick it out in the main hall upside down. It'll be repaired/replaced eventually.

1.2.4 Doors

The lab should be kept secure at all times. This means that the door should only be propped open if there is someone (awake) in the lab. Whenever you leave please check that you are not the last one leaving for the night. It annoys the hardware people to come in for work in the morning, only to find the train lab wide open to the world and unattended. You shouldn't expect any sympathy if machines are stolen due to the carelessness of the people who need them.

1.2.5 Demonstrations

Occasionally there will be tours running through the lab. This is usually only a minor inconvenience and sometimes is a good opportunity for you to appear on TV or in a newspaper. Tour times will be posted on the blackboard or, in the case of really important visitors (i.e., those with money) in the newsgroup. During tour times, the EOS systems will not be available for use.

1.3 Lab etiquette

Try to be thoughtful of others. Nothing more needs to be said here. Keep in mind your classmates are trying to meet the same deadlines you are: don't hog resources.

1.4 Getting started with the software

1.4.1 Setting up your undergrad account

Much of the information you will need to write your programs has been provided on-line in the form of man-pages. To read the pages, and to use the special software, you must configure your account. The minimum required is as follows:

```
setenv PATH `/bin/showpath u=m /u/cs452/i586-3.3.3/bin current`
setenv MANPATH `/bin/showpath c=m /u/cs452/i586-3.3.3/man standard`
```

These two lines could be added to your `.cshrc` file. You should change the lines of similar form that you find there already. However, if you intend to compile 'C' or 'C++' programs for other courses (or personal use), you won't want this environment to be loaded all the time. In this case, put the above two lines into a file and `source(1)` the file when you want to switch to CS452/CS652 work.

With you environment configured properly, you can view man pages for the course as if they were normal man pages:

```
% man cs452
```

This command should show you a page that describes other man-pages related to CS452/CS652.

Chapter 2

Hardware information

2.1 Eos system

Eos is the name given to the computer systems used to teach CS452/CS652. Each Eos system is a common PC clone system assembled from off-the-self components. In fact the parts were chosen so that replacement parts would be available well into the future.

Eos is the Greek goddess of dawn. Since "Eos" is not an acronym, it should not be written in all caps.

The following sections serve to introduce the hardware of the Eos systems. Many of the specifications given here can not be easily discovered otherwise.

Because these computers are PC clones, there is a great deal of public-domain programming information. Some of this information might be applicable to these machines, but you must remember that we do not run MS-DOS on these machines, so any services normally provided by DOS are not available unless you provide them. Also, since the machine is run in protected mode (see the "Introduction to *i486*" section) the services normally available from the BIOS (Basic I/O system) are unavailable. The BIOS is only used to perform power-on self tests (POST) and system initialization.

2.2 Configuration of system

Each machine is configured as follows.

Processor: Advanced Micro Devices AMD486DX2 CPU running at 40Mhz externally (80Mhz internal) with integral FPU.

Memory: 4 meg of RAM, 16k of EPROM, 114 bytes of NVRAM.

I/O Facilities:

- Four 16550 serial ports (three usable).
- Standard IBM-PC keyboard.
- Parallel port.
- Joystick/game port.

Graphics: ATI Mach32 video card with maximum resolution: 640x480 (24 bits colour). Hardware acceleration is available for line drawing, clipping and block move.

Sound: Sound Blaster16 sound card: FM Synthesis (3 channels), stereo ADC and DAC at 44kHz, 16 bits per sample.

Network: SMC Elite16 ethernet card: standard 10Mbps ethernet support.

2.3 Interrupt numbers

An "interrupt number" is an internal number. From the CPU's point of view it's the index into the IDT—the Interrupt Descriptor Table. "IRQ numbers" are the numbers which people normally talk about when dealing with PC hardware. IRQ stands for Interrupt Request and refers to a physical wire on the backplane bus. Each device that wishes to generate an interrupt must be assigned an IRQ.

All 16 IRQ numbers are mapped into the range 32–47 of the interrupt numbers. Interrupt numbers below 32 are reserved by Intel for their use. You may define your own meaning for those 48 and above, since they are generated by software only.

External hardware interrupts (numbered from 32 to 47) are controlled by the ICUs (Interrupt Control Units: the i8259 chip). It is the ICU that manages the hardware IRQ lines and converts them to interrupt requests to the CPU. To get interrupts 32–47, you must unmask (i.e., turn-on, or enable) the interrupt in the ICU and get the hardware to generate the IRQ[1]. How to enable the IRQ varies from device to device. Some always generate interrupts, like the keyboard controller—it does not need to be enabled. Other chips, like the USARTs, allow you to select what serial events will generate an IRQ (but there is only one interrupt line for each port, so if you ask for two interrupt sources, they are combined and you must poll in your handler).

The CPU by default ignores all interrupts (except NMI, which is "not maskable" so it can't be ignored). To enable interrupts, you must execute the instruction `STI`[2] which changes a bit in the processor status word. Once you've done this, interrupts from 0-31 might occur. Next, you must enable the i8259 in question to allow it to pass-on a particular IRQ. Then, you have to get the device to generate an interrupt. For the keyboard, it will always generate an interrupt when a key is pressed. For the serial ports, you must configure what event you want to cause an interrupt in the chip. Most people configure the chip to generate an interrupt whenever a character is received and whenever a character has been transmitted.

In addition to all that work, you also have to configure the i8259s so that they perform the mapping. Plus, the CPU uses a table, called the IDT to decide what to do when an interrupt occurs. However, much of that work has been done for you: You just have to call the library function `loadIdt()` found in examples/useful/idt.cc. This function call sets every interrupt to point to a default handler which prints out a message (and halts the machine). See the next chapter on software for more details, especially Section 3.4.

Some interesting interrupt numbers are as shown in Table 2.1 (after calling `loadIdt()`).

2.4 I/O Port Map

Most hardware devices on the EOS machine are I/O mapped. That is, their control registers appear in a separate address space dedicated to I/O. Access to this "other" address space requires special I/O instructions (see header file `<machine/pio.h>`). In particular, the functions listed in Table 2.2 can be used to access I/O space.

The header file `<machine/pc/isareg.h>` defines most of the port numbers discussed here.

2.4.1 Serial ports (NS16550)

Port	Number	Comments
0x03F8	0	Connected to train track or robot.
0x02F8	1	Connected to debug terminal (console).
0x03E8	2	Connected to an X-terminal (remote debug).
0x02E8	3	Unusable, conflicts with 8514/A compatibility.

Each serial port operates identically and consists of a number of registers. Each register has its own I/O port, the numbers above are just the base address of each set. The registers are summarized in Table 2.3[3].

To gain access to a particular register of a serial port, take the base I/O port address and add the offset of the register in question. Thus, to read the "Line status register" of port one, the command "`inb(0x02fd)`"

[1] IRQs 0–7 going to the first ICU are mapped to interrupt numbers 32–39, while IRQs 0–7 going to the second ICU are numbered 40–47 as interrupts.

[2] A macro is defined in `<machine/cpufunc.h>` to include this op-code into a 'C' program.

[3] "DLAB" is bit 7 of the LCR. When set, it changes the function of the first two registers.

Number	Suggested name	Comments
0	INT_DIVIDE_EXCEPT	Occurs after a divide by zero
1	INT_DEBUG_EXCEPT	Results from single-stepping
2	INT_NMI	Non Maskable Interrupt: after button is pressed
3	INT_BREAKPOINT	Opcode 0xCC generates this
4	INT_OVERFLOW_EXCEPT	An exception: numeric overflow
6	INT_OPCODE_EXCEPT	Bad opcode exception
8	INT_DOUBLE_EXCEPT	Two exceptions occurred "simultaneously"
12	INT_STACK_EXCEPT	Trouble with the stack
13	INT_GP_EXCEPT	GPF: General Protection fault, the catch-all exception
others		See *i486* data book for other internal exceptions/interrupts.
(32+0)	INT_TIMER	(IRQ0 to first 8259) Interval timer interrupt
(32+1)	INT_KEYBOARD	(IRQ1 to first 8259) Keyboard (key pressed)
(32+3)	INT_SERIAL1	(IRQ3 to first 8259) Serial port 1 event
(32+4)	INT_SERIAL0	(IRQ4 to first 8259) Serial port 0 event
(32+5)	INT_SOUND	(IRQ5 to first 8259) Sound event (end of DMA)
(32+8)	INT_RTC	(IRQ0 to second 8259) Real-time clock tick
others		Ethernet card, vertical refresh, parallel port, etc.

Table 2.1: Interrupt numbers

Function	Description
`u_char inb(u_short port)`	Input a single byte from the given port.
`void insb(u_short port, void *addr, int cnt)`	Input a block of bytes from the given port.
`u_short inw(u_short port)`	Input a single word (2 bytes) from the given port.
`void insw(u_short port, void *addr, int cnt)`	Input a block of words (2 bytes each) from the given port.
`void outb(u_short port, u_char data)`	Output a single byte to the given port.
`void outsb(u_short port, void *addr, int cnt)`	Output a block of bytes to the given port.
`void outw(u_short port, u_short data)`	Output a single word (2 bytes) to the given port.
`void outsw(u_short port, void *addr, int cnt)`	Output a block of words to the given port.

Table 2.2: Interesting functions defined in <`machine/pio.h`>.

Offset	Mode	Name	Function (see data sheet for more information)
0x0	w	THR	Transmitter holding register
0x0	r	RBR	Receiver buffer register
	r/w	DLLB	Divisor latch, low byte (when DLAB=1)
0x1	r/w	DLHB	Divisor latch, high byte (when DLAB=1)
	r/w	IER	Interrupt enable register (when DLAB=0)
0x2	r	IIR	Interrupt identification register
	w	FCR	FIFO control register
0x3	r/w	LCR	Line control register
0x4	r/w	MCR	Modem control register
0x5	r	LSR	Line status register
0x6	r/w		Scratch register

Table 2.3: Serial port chip registers.

could be used. However, all the useful values have been defined in the header file <`machine/serial.h`>. Therefore, you can also write the above as:

```
#include <machine/serial.h>
#include <machine/pio.h>

char c = inb(USART_1_BASE + USART_LSR);
```

For more information on programming the serial ports, please consult the data sheets. The serial ports are implemented using an NS16550. This is an improved version of the Intel chip, i8250. The only addition is the on-chip FIFO and the NS16550 should be software-compatible with i8250 drivers.

2.4.2 Timer (PIT)

The programmable interval timer is implemented with a very limited Intel chip, i8253. It has three independent counters. Each is 16 bits wide and the whole chip is clocked with just one clock frequency. There is no way (in hardware) to chain or cascade these counters.

The output of counter zero is connected to IRQ0, so it can generate an interrupt when terminal count is reached. Counter 1 is available, but its output pin is not connected to anything useful (DRAM refresh is done in dedicated hardware now). Counter 2 is used to generate square waves for the tiny speaker inside the case. Its output is gated by a bit in the PPI chip (keyboard controller).

Port	Mode	Description
0x0040	r/w	Counter 0, counter divisor
0x0041	r/w	Counter 1, RAM refresh counter (unused)
0x0042	r/w	Counter 2, speaker tone control
0x0043	r/w	Mode port, control word register for counters 0-2

The bits of the mode port are as follows.

Bits	Value	Meaning
bit 7-6	00	counter 0 select
	01	counter 1 select
	10	counter 2 select
bit 5-4	00	counter latch command
	01	read/write counter bits 0-7 only
	10	read/write counter bits 8-15 only
	11	read/write counter bits 0-7 first, then 8-15
bit 3-1	000	mode 0 select
	001	mode 1 select - programmable one shot
	x10	mode 2 select - rate generator
	x11	mode 3 select - square wave generator
	100	mode 4 select - software triggered strobe
	101	mode 5 select - hardware triggered strobe
bit 0	0	binary counter 16 bits
	1	BCD counter

A NetBSD header file has a good description of how to interact with this chip.

> There are three mode registers and three countdown registers. The countdown registers are addressed directly, via the first three I/O ports. The three mode registers are accessed via the fourth I/O port, with two bits in the mode byte indicating the register. (Why are hardware interfaces always so braindead?).
>
> To write a value into the countdown register, the mode register is first programmed with a command indicating which byte of the two byte register is to be modified. The three possibilities are load msb (TMR_MR_MSB), load lsb (TMR_MR_LSB), or load lsb then msb (TMR_MR_BOTH).

To read the current value ("on the fly") from the countdown register, you write a "latch" command into the mode register, then read the stable value from the corresponding I/O port. For example, you write TMR_MR_LATCH into the corresponding mode register. Presumably, after doing this, a write operation to the I/O port would result in undefined behavior (but hopefully not fry the chip). Reading in this manner has no side effects.

The header file `<machine/pc/timerreg.h>` has registers values defined for this chip (including those mentioned above). Please see the data sheets for more details on this chip.

The frequency supplied to this chip is fixed at 1,193,182Hz. This means if you use $2^{16} - 1$ (0xffff) as a divisor, you'll get interrupts at a rate of 18.2Hz, just as MS-DOS does.

The following code sets the PIT to cause an interrupt every 20^{th} of a second (20Hz).

```
#include <machine/pc/timerreg.h>

outb(TIMER_MODE, TIMER_SEL0|TIMER_RATEGEN|TIMER_16BIT);
outb(IO_TIMER1, TIMER_DIV(20)%256);
outb(IO_TIMER1, TIMER_DIV(20)/256);
```

2.4.3 Real time clock (RTC)

The "real time clock" on PC-compatible machines is meant to provide time and date functions—it is not a high-resolution or high-precision timer. The same chip, since it must maintain its time with the power off, also carries a small amount of battery-backed SRAM. Configuration values can be stored when the power is off in this memory. The version of this chip installed on Eos machines is the DS12887 which provides 114 bytes of NVRAM[4]. Please do *not* change the values in the NVRAM. (It is easy to fix them when they are wrong, but it is a nuisance.)

Ports 0x070 and 0x071 are used to access this chip. The first port (0x070) is an address port, and 0x071 serves as a data port. To access an address (one of 128) inside the RTC, first output the index (i.e., address) to port 0x070, then read or write the desired data from port 0x071. You must be careful to always follow a write to 0x070 with some action on 0x071, or the RTC will be left in an unknown state.

Table 2.4 lists each important address (index) and its function. For the exact meaning of all these registers, consult the data sheets. Some useful functions and defines can be found in `<machine/clock.h>`.

The high bit (7) of the RTC address port seems to gate the NMI input to the CPU, but there is no good documentation on this. Normally it is left clear. The interrupt output of the RTC is connected to IRQ0 of the second ICU (i.e., IRQ8) so it can be used to generate periodic interrupts. These interrupts can be configured to occur at a much higher rate than is possible with the PIT. In fact, with careful programming, it is possible to accurately measure times with microsecond resolution using this chip.

Please see the software section for a discussion of which timer chip (RTC or PIT) is the best for your kernel.

2.4.4 Interrupt Control Unit (ICU)

All hardware interrupts on an IBM PC are funneled through a chip called variously, the ICU (Interrupt Control Unit) or PIC (Programmable Interrupt Controller). The design is based on the Intel i8259. The function of this chip is to simplify the interrupts that the CPU sees. For example, you can mask (turn off) interrupts from a given device using the ICU. Also, the ICU prioritizes interrupts so that higher-priority devices will preempt lower-priority ones.

The first ICU is located at 0x020. The secondary (slave) ICU is located at 0x0a0. The first ICU controls 8 IRQs, one of which (IRQ2) is an IRQ from the secondary ICU, which controls another 8 IRQs. When the secondary ICU needs to cause an interrupt, it generates an interrupt for the other ICU. Their interaction is not quite as simple as that, however.

[4]PC users call this "the CMOS memory", since they believe it is the only chip in the machine made using the mystical fabrication technology of CMOS.

Index	Description
0x00	current second in BCD
0x01	alarm second in BCD
0x02	current minute in BCD
0x03	alarm minute in BCD
0x04	current hour in BCD
0x05	alarm hour in BCD
0x06	day of week in BCD
0x07	day of month in BCD
0x08	month in BCD
0x09	year in BCD (00-99)
0x0A	status register A
	bit 7 = 1 update in progress
	bit 6-4 divider that identifies the time-based frequency
	bit 3-0 rate selection output frequency and int. rate
0x0B	status register B
	bit 7 = 0 run
	= 1 halt
	bit 6 = 1 enable periodic interrupt
	bit 5 = 1 enable alarm interrupt
	bit 4 = 1 enable update-ended interrupt
	bit 3 = 1 enable square wave interrupt
	bit 2 = 1 calendar is in binary format
	= 0 calendar is in BCD format
	bit 1 = 1 24-hour mode
	= 0 12-hour mode
	bit 0 = 1 enable daylight savings time. Only in USA, useless in Europe.
0x0C	status register C
	bit 7 = interrupt request flag
	bit 6 = periodic interrupt flag
	bit 5 = alarm interrupt flag
	bit 4 = update interrupt flag
	bit 3-0 reserved
0x0D	status register D
	bit 7 = 1 Real-Time Clock has power
	bit 6-0 reserved

Table 2.4: Real-time clock registers.

```
#include <machine/pc/isareg.h>
#include <machine/pio.h>

inline void maskIRQ(uchar irq, bool flag)
{
    uint    port = IO_ICU1+1;
    uchar   val;

    if(irq >= 8) {        /* an IRQ on the secondary ICU */
        port = IO_ICU2+1;
        irq -= 8;
    }

    val = inb(port);

    if(flag) val |= (1<<irq);
    else     val &= ~(1<<irq);

    outb(port, val);      /* mask the interrupt enable */
}
```

Figure 2.1: Code for masking/unmasking IRQs in the ICUs.

This chip seems to be the oldest, most primitive component in a typical PC. It has a lot to do with why PCs are so evil. In particular, it is extremely difficult to understand its documentation. What follows is a simplification, and I cannot recommend learning more about this chip.

The file examples/useful/idt.cc contains a function `loadIdt()`, which initializes and configures the i8259 for you. You may want to examine the code it contains, but I don't recommend changing how it works. After you have called that code, your interaction with the ICU can be limited to masking and unmasking individual interrupts, and sending EOI (End Of Interrupt).

Each IRQ can be enabled or disabled individually. The code in `loadIdt()` disables all the IRQs on both chips. Therefore, once your have a device generating IRQs (interrupt requests), you must unmask that device's IRQ line in the ICU. You can do that with the function listed in Figure 2.1.

A `flag` value of one *disables* the IRQ, while zero *enables* it. The values `IO_ICU1` and `IO_ICU2` are defined in <`machine/pc/isareg.h`> as 0x020 and 0x0a0 respectively.

When an IRQ occurs, the ICU prioritizes it and decides (if that IRQ is unmasked) to signal the CPU. The ICU must be notified when the processing for that interrupt is complete, so that it knows when it is safe to interrupt the processor again. The signal to the ICU that does this is EOI. You *must* tell the ICU that the interrupt handling is complete, or else you will never get an interrupt again from that device (IRQs at higher priority will still occur). Use the following code to signal EOI.

```
outb(IO_ICU1, 0x020);         /* non-specific EOI */
outb(IO_ICU2, 0x020);         /* non-specific EOI */
```

This code will acknowledge whatever the last interrupt was from the ICU. It is possible to acknowledge individual IRQs (i.e., to service interrupts out of order) but you don't need to do that. To simplify things, this code sends an EOI to both ICUs, so that the same code can be used for IRQs from either chip. Technically, you shouldn't send an EOI to the secondary ICU if the IRQ was caused from an IRQ handled from the first ICU, but it doesn't cause any trouble.

2.4.5 Keyboard Controller (PPI)

The keyboard is controlled by an Intel i8042 chip. This chip is normally used for a Programmable Parallel Interface (PPI) but has be used over the years to add kludges to the PC for enhancements. Since you normally will never need to program this chip, it will not be extensively documented here.

The keyboard controller is located at 0x060 to 0x064 in I/O address space. Appendix **??** lists the control registers and their usage. If you want to read the PC keyboard, use the functions provided in examples/useful/kbd.cc.

Name	Part #	#	IRQ	I/O Base	Function
Serial	NS16550	0	4	0x3f8	Provides serial port (to train).
		1	3	0x2f8	(to serial console)
		2		0x3e8	(for remote debug)
PIT	i8253		0	0x040	Simple countdown timers (3).
RTC	DS12887		8	0x070	Time/date functions, periodic interrupt.
ICU	i8259	0	*	0x020	Interrupt controller, manages interrupt sources.
		1	2	0x0a0	
PPI			1	0x060	Keyboard controller and kludge corner.

Table 2.5: Location and function of all chips.

2.5 Memory map

Figure 2.2 is a map of memory on the EOS system. This is the state of things when your program begins.

As you can see, most of the system RAM is continuous, except for a small amount at 0x8,0000 to 0x10,0000. Your kernel will be loaded by grub at the one meg boundary (0x10,0000). Thus, the maximum size for a kernel and its modules is 3 megabytes. If more memory is added to the systems in the future, then it will appear from 0x40,0000 upwards.

The video memory at 2gig is not mapped until certain functions in the graphics library initialize the video card. There is no need to access the "video aperture" directly since library functions are provided to manage it. See `graphics(3)` and `libsvga(3)`.

The addresses that appear here are called *linear addresses* or *physical addresses*. Most programs cannot convert linear addresses into pointers because the typical program does not know where in memory it is located. All logical addresses are relative to the base of a certain segment. That segment might be located anywhere in the address space. Without access to the GDT it is not possible to determine where a segment is located in memory.

You probably don't want to use the memory between 0x400 to 0x500 because the machine will complain when it is rebooted (soft reset only). The BIOS work area is examined (and used) during boot.

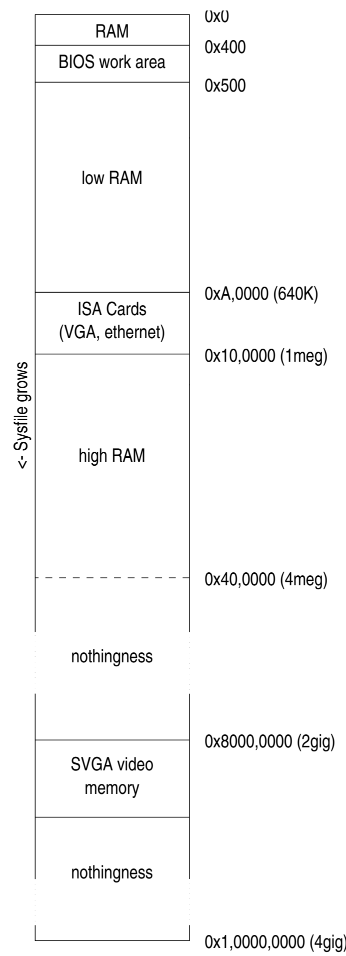


Figure 2.2: Memory map.

2.6 Summary: What chips do what

Please see Table 2.5 for a complete list of all the important chips. The data sheets which are attached to this manual (and available in the lab) give documentation for most of the chips listed.

2.7 Miscellaneous hardware items

- The NMI button installed on each machine can only be used once per reboot. For some reason, the hardware latches the first usage, so that pressing it again will have no effect. There may be some way to reset this in software (in your NMI handler), but it has never been found.

Chapter 3

Software information

3.1 Introduction to the *i486*

3.1.1 History

The *i486* processor comes from a long line of processors which can be traced back to the first microprocessor. This history starts in 1972[1] with the 4004. The 4004 was designed for a (then) powerful calculator. It was a 4-bit processor, which means that its registers were 4 bits wide. The 4004 was followed by the 8008 which was an 8-bit version.

As an upgrade to 8008, the 8080 was released in the mid-seventies. It was very popular and many CP/M systems were built around it and its clones. The Zilog Z80 series, which is an enhanced clone chip, is still popular today. The first PC was built on the the 8088 chip[2]. This chip was designed to ease transition from the 8080. It had special features such as a half-carry flag which would allow machine language programs to be more easily ported. The 8088 is not very well designed, but it was advanced for its time and allowed the mixing of 8-bit programs and 16-bit programs by using segments.

On an 8-bit processor, you could only access 64k directly. One word would be 16-bits and require two registers. The new processor from Intel keep this 16-bit pointer model model but added the concept of segments. With the 8088, you can only address memory indirectly via a segment register. The segment register defines a base address, and the 16-bit pointer value is added to the base to yield a longer address. Unfortunately, they were foolish and only allocated 16-bits to the segment base address and made its granularity 16 bytes (one paragraph).

Therefore, a segment register would point to the start a paragraph boundary in memory, and all references would use a segment offset (pointer) which was 16-bits long. For all this effort, they had only increased addressable memory to one megabyte! On an 8088 processor there are four segment registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), and ES (Extra Segment). Each segment register is dedicated to the purpose from which it gets it name, because each instruction implicitly refers to an segment. For example, if you jump to an address in assembly language, that address is always relative to the CS (Code Segment). At the added cost of one byte, any instruction could use a "segment override prefix" so that its operands would reference some non-default segment.

The rationale behind this is that most programs have pointers with an implicit type: labels into code or pointers into memory. So the obvious thing is to split the two parts into a code segment and a data segment. The program can even be relocated just by changing the segment bases. Well-behaved programs wouldn't change their segment bases and would thus work anywhere in memory, and they could access $3 \times 64k = 192k$ of memory directly and naturally.

Unfortunately, the IBM PC was a hit and took over the world. The 8088/8086 family was renamed and upgraded through the parts: i186, i286, i386, i486 and the Pentium. When the i286 was introduced a

[1] Some of the information in this section was culled from http://bwrc.eecs.berkeley.edu/CIC/archive/cpu_history.html, which is a great read.

[2] Actually, the PC was originally designed with the superior MC68000, but IBM's fabrication facilities could not construct the machines, so it was entirely redesigned for Intel chips instead. The MC68000 was too large, because it had a simpler bus interface and hence more pins.

limited "protected mode" feature was added that fixed some of the problems with segments but it was poorly conceived and it was is difficult to create a real multi-user system with the i286. The i386 had a better implementation of protected mode and its programmer's model has been carried all the way to the Pentium.

The main feature of protected mode is that it allows hardware memory protection and extends access to a full 32-bit address space. In protected mode, even an Intel processor looks reasonably modern.

Because of the similarities between the generations, any i386 plus reference can be used be used for this course, since there have been no (significant) visible changes between the i386 and i486.

3.1.2 Programmer's model

The *i486* has 8 relatively general-purpose registers. Each is one word (32 bits) wide. There are 6 segment registers, one processor status word and one instruction pointer (or program counter, if you prefer). Please see Figure 3.1 for a graphical representation. As you can see, there are aliases for some of the byte/half word components of the registers.

In addition to the "general purpose" registers, there are segment registers, named: CS (Code Segment), SS (Stack Segment), DS (Data Segment), ES (Extra Segment), FS (Data Segment) and GS (Data Segment).

The layout of a running program on an EOS system is as shown in Figure 3.3.

Your programs will all run in "protected mode". In this mode, all access to memory is done indirectly through selectors. A selector is a 16-bit value that is an index into the GDT (Global Descriptor Table). In the GDT, the entry for a selector has the base address (physical, 32-bit value) and limit (or length) of the segment. The GDT entry also holds privilege information and some other values. I will use segment and selector interchangeably, except a selector might represent an object other than a block of memory. For example, a TSS (Task State Segment) is a selector, but it actually points to an internal table used for multitasking.

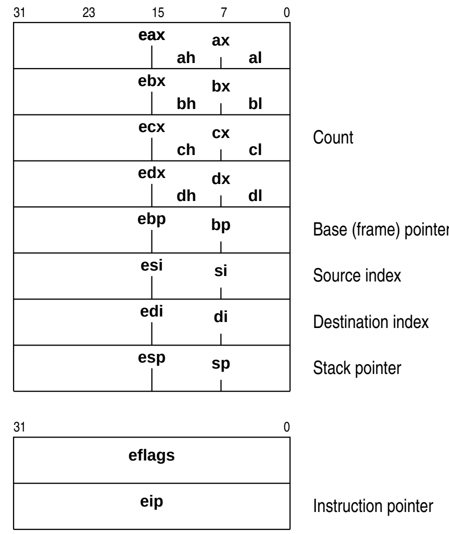


Figure 3.1: Registers on the *i486*.

The *i486* implements a simple protection model. There are four privilege levels: 0, 1, 2 and 3. Theses are usually called rings, because they are often represented as concentric circles. The inner-most circle is ring 0 which can do anything (the kernel). Ring 1, 2, and 3 can each do less. User tasks should be implemented at ring 3 so that they cannot interfere with anything else[3]. The current PL (Privilege Level) is stored in the processor status word (EFLAGS). Whenever a selector is loaded into a segment register, the access is checked: the PL of the selector (stored in the DPL field of the GDT entry) must be numerically greater or equal to the PL of the running processor. This means that no task can access a selector of lower PL. Unfortunately, to switch PLs, you must use the *i486*'s multitasking features. They are very difficult to use, so I will assume your system will run entirely at ring 0. Therefore, you should use zero everywhere a PL is required.

One task that your kernel must perform is GDT management. Whenever a new segment is to be used (perhaps to start a new task), a new entry in the GDT must be made. See Section 3.3 for more details and advice concerning the GDT.

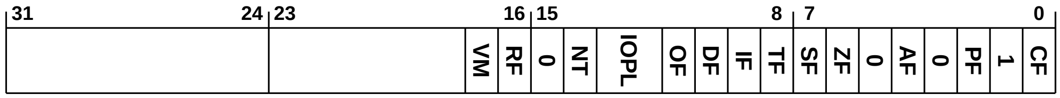
3.1.3 Register values

During the execution of a program on the EOS system, each register has a specialized purpose.

The registers EAX, EBX, ECX and EDX are available for general calculation. However, if you examine the instruction set closely, you will see that some of these registers are used in special ways by special instructions. For example, some instructions require (or produce) a 64-bit value and this value will typically be stored in two, hard-coded, registers. EAX is used to return values from 'C' functions. Arguments to functions are passed on the stack.

ESP is the stack pointer. It points to the top of the stack. The stack on the *i486* grows downward, towards lower values. The value at SS:ESP is the current value on the top of the stack. Therefore, to simulate a push operation, you should subtract 4 from ESP and *then* write the new value to SS:ESP. A pop is equivalent to returning the contents of SS:ESP and *then* adding 4 to ESP. This is all assuming that your

[3] Some systems use ring 1 for device-drivers, but there is little benefit to such a system, because switching rings is very expensive (time) and the protection gained is slight.



CF	Carry flag	S	PF	Parity flag	S	AF	Auxiliary carry	S
ZF	Zero flag	S	SF	Sign flag	S	TF	Trap flag	S
IF	Interrupt enable	X	DF	Direction flag	C	OF	Overflow	S
IOPL	I/O privilege level	X	NT	Nested task flag	X	RF	Resume flag	X
VM	Virtual 8086 mode	X						

Note: S – Status flag, C – Control flag, X – System flag.

Figure 3.2: The EFLAGS register definition.

are pushing/popping full words (4 bytes, 32 bits) from the stack. You should never need to do otherwise, since all values on the stack are word aligned in protected mode.

EBP is the base pointer. It is used by GCC to allow debugging. At the start of every function, it is loaded with the initial value of ESP (the stack pointer). The debugger will use this value to find the local variables of the current function. You can use the option `-fomit-frame-pointer` to GCC to suppress this behaviour, but the resulting program cannot be easily debugged. On real computers, this register would be called the frame pointer, because it points to the current call frame. Function arguments can be referenced as negative offsets to the EBP, while function local variables can be referenced as positive offsets to EBP.

EDI and ESI are pointers used during certain copy instructions. They can be used for other purposes and GCC does use them as general registers.

EIP is the instruction pointer. You cannot use normal move commands to put values into it. Instead, you must use a branching statement, jump or call instruction. It always points to the next instruction, not the current instruction.

EFLAGS is an important register. It contains much of the state of the processor. Figure 3.2 shows the meaning of each of the bits.

The processor also has four control registers, called CR0 through CR3, but you should not need to change these. They are used to switch into protected mode and to enable/handle paging.

3.1.4 Assertions you should maintain

At all times, the CS must point to executable code. The other segment registers should all be set to point to the data segment. The only exception to this is ES, FS, and GS which can be temporarily used for other segments (or set to zero). However, protection is checked whenever a segment is loaded, so you cannot store general values in a segment register or else you will get a protection fault (exception 13: GPF, General Protection Fault).

The layout of a running program on an EOS system is as shown in Figure 3.3. As operating system implementors, it is your job to make sure that the situation depicted in the diagram is never violated (except briefly, perhaps). The CS must point to the first byte of the `elf` program's code segment. The DS must point to the first byte of space to be used as the program's data space. The compiler generates code on the assumption that DS=SS. In particular, if you take the address of a local variable (one that is allocated on the stack), then for that pointer to be valid (i.e., equivalent to a pointer to a global variable), then DS must equal SS.

The code segment and the data segment can be anywhere in the address space. However, CS will normally point to a selector whose base address is somewhere inside the kernel or module loaded into memory by Grub. `crt0.o(3)` will normally assume that CS and/or DS is big enough enough to allow access to the entire elf file. It also normally assumes that when the program is started, DS points to a data segment which can be used. ESP should already point to the end of the data segment (the bottom of the stack) when the program is first started.

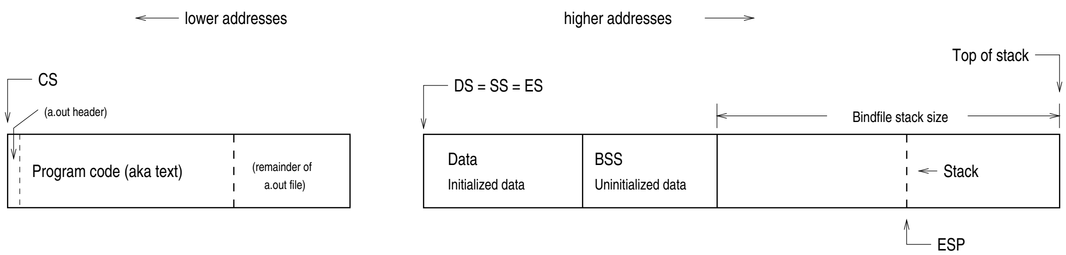


Figure 3.3: A running program on an EOS system.

3.2 Suggestions on structuring your kernel

3.2.1 The KISS principal

By now, you should have been exposed to the KISS principal: Keep It Simple, Stupid. The *i486* is a very powerful processor and it is tempting to try and exploit all of its features. Don't do it. You will have enough trouble trying to make a working kernel as it is. Don't be distracted by cool features of the hardware. You must start by getting something simple working. When you have the basics working, you can move onto more complex things. A working, but simple kernel, is much more impressive than a design for a neato kernel that will never be working.

3.2.2 Kernel design suggestions

Here are some strong suggestions on how to structure your kernel. You may not have time to finish your assignments if you don't implement things this way.

- Everything should run at ring zero. Do not use the TSS features.
- The GDT should limit the size of data segments to exactly the expected value. Code segments must be read-only and limited to the size of `ELF` module (not the size of the text section).
- Nothing should be written in assembler except things that setup context for C. That is, actual code to handle an interrupt should be C, but the first part of the ISR needs to be assembler.
- The idle task should not consume more of your attention than the final application.

Here are some weak suggestions on how to structure your kernel.

- Handling an interrupt (that doesn't cause a re-schedule) should be faster than switching between tasks.
- The kernel should have a "high" and a "low" part. The high part has a full processor context (just as much as a regular process) and the low part has only a partial context and is only used for processing hardware interrupts (and maybe scheduling).
- There should be a single entry point to the kernel, and a generalized kernel call interface.
- Interrupts should be handled by specialized entry point(s) into the kernel.
- Optimize your design for speed, except when it adds serious code complexity.

3.2.3 Why hardware memory protection is hard to implement

It is extremely hard to implement hardware memory protection on the *i486* because you must use TSS entries in the GDT to do it. There is *no way* to debug your usage of the TSS. If even one bit of the TSS is wrong, you'll get a GPF. If you're lucky you might get a more informative exception, but there is a long list of reasons why the processor might object to your TSS entry and most of them result in a GPF. It is a great deal of work to get the TSS entry correct and trial-and-error will likely fail.

If you set limits to the segments you create in your GDT, you will still get more protection from pointer mistakes than students in previous years have gotten. The only thing you would learn from mastering the TSS is obscure *i486* knowledge. Unless you intend to become a *Microserf*, it won't be useful information.

3.3 GDT: Global Descriptor Table

3.3.1 Introduction

The GDT (Global Descriptor Table) is an important table used in protected mode on the *i486*. A selector value is an index into this table (sic, see below), so all selectors[4] must have an entry in the GDT. Each entry in the GDT defines a number of things for a segment: where it is located in physical memory, how long the segment is, whether it is executable, and a number of privilege details.

Each entry is 8 bytes long. The first entry of the GDT is unused. The header file <`machine/segments.h`>, defines a useful structure for working for GDT entries:

```
struct segment_descriptor {
    unsigned sd_lolimit:16;         /* segment extent (lsb) */
    unsigned sd_lobase:24;          /* segment base address (lsb) */
    unsigned sd_type:5;             /* segment type */
    unsigned sd_dpl:2;              /* segment descriptor priority level */
    unsigned sd_p:1;                /* segment descriptor present */
    unsigned sd_hilimit:4;          /* segment extent (msb) */
    unsigned sd_xx:2;               /* unused */
    unsigned sd_def32:1;            /* default 32 vs 16 bit size */
    unsigned sd_gran:1;             /* limit granularity (byte/page) */
    unsigned sd_hibase:8;           /* segment base address (msb) */
};
```

You can read your *i486* manual for the exact meaning of all the fields, but you need only concern yourself with the following fields:

base The first byte of the segment, in physical memory, is defined by the 32 bits composed of:

```
(sd_hibase<<24) | sd_lobase
```

Your segments may begin anywhere in memory.

granularity The `sd_gran` flag sets the granularity of the limit field. Since only 20 bits are reserved for the limit, you can only define segments greater than 1M if the granularity bit is true. When true, the granularity of the limit field is 4k pages, otherwise, it is measured in bytes.

limit The maximum address (plus one) of any reference via a selector is set by the limit field. It is composed from: `sd_lolimit | (sd_hilimit << 16)`.

type The `type` of the selector for your applications, should be one of two values: 0x01f (for code segments), or 0x013 (for R/W data segments).

others The other values should be defined as follows: `sd_dpl = 0`, `sd_p = 1`, `sd_def32 = 1`.

Selector values are not really indices into the GDT, but are actually a structure. The top 13 bits are the selector index in the GDT, and the lower three bits are a bit (table indicator) and a PL (requestor's privilege level). However, since everything is at ring zero, these lower three bits will always be zero. Since each GDT entry is 8 bytes long and $2^3 = 8$, the selector can be treated as a pointer into the GDT, as long as its lower bits are zero.

[4]There are local selectors, defined only in the LDT (Local Descriptor Table) that won't be discussed here.

3.3.2 Initial values, expected values for libraries

Once your kernel is running, you are no longer guaranteed that a GDT exists. Grub will of course create a GDT when switching into protected mode and booting your kernel, but the Multiboot standard states that the GDT is undefined on kernel startup. You will have to create your own GDT as soon as possible in your kernel. This includes adding entries for your already running kernel.

If you use the library routines: `absZero()`, `absRead()` and `absWrite()`, found in examples/useful/abscopy.S, you must be aware that they're written to use selector 0x10, set up to provide access to the entire memory space.

3.3.3 Suggested implementation

The *i486* processor stores the location of the GDT in a special purpose register called GDTR (GDT Register). This register contains both the base address and length of the GDT. To load these values, you must construct a structure of type `region_descriptor` (defined in `<machine/segments.h>`) and execute the `LGDT` (load GDTR) instruction. An interface to this instruction is provided in `<machine/cpufunc.h>`. To read where the GDT is currently located, use the `SGDT` (store GDTR) instruction. Individual records in the GDT are read when the selector is first loaded into a segment register. Any exceptions caused by bad values will occur at that time. It is safe to define a GDT that is the maximum size (8192 entries, or 64k bytes) without defining the unused values.

To establish your own GDT you must allocate your own array. This will put it in the kernel's data segment and make access to the GDT as easy as any array reference.

In summary, here are the (recommended) steps to initializing your own GDT.

1. Declare an array of `struct segment_descriptor` big enough for your long-term needs.

2. Set the GDTR to point to your array. Calculate the base address of your table in physical memory. Use `sizeof()` to determine the length.

3.4 IDT: Interrupt Descriptor Table

3.4.1 Introduction

The IDT (Interrupt Descriptor Table) is a table like the GDT, but it is used to store information about interrupt and exception handlers. Each entry in the IDT is 8 bytes and defines (among other information) the entry point of the ISR (Interrupt Service Routine) used to handle a given interrupt.

Please see Section 2.3 for a complete listing of interrupt numbers and further discussion.

3.4.2 Using library calls for it

You are not expected to understand the IDT. There are library functions in examples/useful/idt.cc for everything you should need. The function `loadIdt()` is used to setup the IDT. Once the IDT has been initialized, you can configure any interrupt to call your ISR with the following code.

```
#include <machine/segments.h>
void isrFunction();
main() {
    loadIdt();
    setIdt(INT_NMI, isrFunction, SDT_SYS386IGT, /*dpl:*/ 0);
}
```

This code will update the IDT, so that next time an NMI occurs your function, `isrFunction()`, will be called. The function given to `setIdt()` is assumed to be located in the current code segment. Your function must save all registers and load all segment registers (except CS). Therefore, the function `isrFunction()`

must be written in assembly language. Typically, the entire ISR will not be written in ASM, only a small stub that establishes the context, and another stub that restores the old context. It is important to preserve the context of the machine correctly when handling an interrupt, or else your system will be unreliable. In particular, you must save all registers, including the segment registers. It is best if you don't interpret the values (i.e., you know what task is running, so you can infer DS, CS values; but don't).

3.4.3 How to do interrupts

Here are all the steps involved in enabling and getting an interrupt. This example is for the keyboard.

1. Load the IDT with default values: call `loadIdt()`.
2. Set the entry for INT_KEYBOARD (33) to point to your ISR by calling `setIdt()`.
3. Enable IRQ generation from the device. For the keyboard controller, you don't have to do anything since it always generates interrupts. For the serial ports, you would have to set certain bits in the IER register.
4. UN-mask the interrupt in question at the ICU. To do this, copy the code shown in Figure 2.1. Call `maskIRQ()` with arguments IRQ=1, and flag=0.
5. Set the interrupt enable flag in the processor. You can do this either by moving a new value into EFLAGS or by executing the instruction `STI`.

The order of these steps is fairly important. It is best to do them in this order, so that things are not enabled before they are valid.

The steps in the initial part of an interrupt handler are as follows.

1. Turn off interrupts (disable nesting).
2. Push all general registers.
3. Push all segment registers (or at least: DS, ES).
4. Get access to your kernel's data segment. You can save a copy of the DS value somewhere in the code segment, or use a hard-coded value. Set DS to this value.
5. Save current ESP and maybe SS.
6. Set SS = DS, and in the next instruction, set ESP to a useful value. You can use a temporary stack fairly easily, or you can try to use the kernel's normal stack. You cannot, however, use the interrupted task's stack since DS must equal SS.
7. Call a C function to do all the real work.

Somewhere in your ISR, you must give the EOI signal to the ICU. See Section 2.4.4 for more information. To return from an ISR, you must do the above steps in reverse order.

1. Load former SS:ESP value.
2. Pop segment registers off stack (reverse order).
3. Pop general registers off stack.
4. Perform `IRETL` instruction: return from interrupt.

Since the interrupt instruction saves the EFLAGS on the stack, returning from interrupt will turn on interrupts again when the EFLAGS are popped from the stack.

3.5 Serial ports

3.5.1 Configuration

Please read the data sheets for information on how to program the serial chips. Grub will not initialize the serial ports for you.

You will probably want to leave both ports configured to generate interrupts on transmit and receive all the time. There is no need to disable transmit on these chips.

3.5.2 ISR structure

The ISRs for the serial ports (two in all), should probably have the following form.

1. (All the ASM code described above to establish context for C code.)
2. Poll the chip for source of interrupt. This is done mostly to clear the IIR.
3. Poll the chip to see if a character can be read from it. Read a character from it, if possible. (Loop here if FIFOs are enabled.)
4. If any characters are waiting to be transmitted, and the chip is ready to send, send one character. (Loop if FIFO enabled.)
5. Send EOI to ICU.
6. (Return via ASM stub, see above section.)

When you are sending a string, and the last character has just been sent, you will get one final interrupt after it is transmitted. If you have nothing to do, don't do anything except read the IIR and LSR, and you won't get another interrupt for transmit. When you first need to transmit something, you should "prime" the interrupts by outputting the first character directly (i.e., not in response to an interrupt). Once that character has been sent, the interrupt will be called and your interrupt-time code can handle the rest of the string.

3.5.3 Cautions on the use of FIFOs

You need to give some thought to using the FIFOs (First In First Out) on the serial ports. At first glance, they sound great since they should reduce the interrupt load on your system. This means that you'll have more time for CPU intensive things such as your idle task. However, it is better not to use them because they reduce the responsiveness of your system. In fact, they really cannot be used effectively with the train track.

The effect of FIFOs on receive is that they delay the interrupt until either enough characters (as configured) have arrived or a timeout has occurred. The timeout time is measured in character times, so it is rather long. Since only a few characters (one response) is sent back from the track at one time, it almost never fills the FIFO and therefore, you system sits idle until some time has passed. You cannot use the full bandwidth to the track this way.

3.6 Timer chips

3.6.1 Making the choice: bad or worse

There are two sources of periodic interrupts on the Eos systems: the PIT or the RTC. As you can see from the hardware section on these two chips, neither is very advanced.

You have your choice of which chip to use to generate your timer ticks. The RTC can tick much faster, giving you more resolution on your time values; however, you have only very coarse control over the rate of interrupts. The PIT, on the other hand, can only generate relatively slow interrupts.

RS0–3	**Interrupt Rate**
0	None
1	3.90625 ms
2	7.8125 ms
3	122.070 μs
4	244.141 μs
5	488.281 μs
6	976.5625 μs
7	1.953125 ms
8	3.90625 ms
9	7.8125 ms
10	15.625 ms
11	31.25 ms
12	62.5 ms
13	125 ms
14	250 ms
15	500 ms

Table 3.1: Interrupt rates available from RTC chip.

The RTC generates IRQ8, which is not the highest priority interrupt. Thus, the serial ports will be serviced before clock ticks (IRQ3 and IRQ4). This means you might lose ticks if the serial port service time exceeds the period of the clock. The PIT is at IRQ0 and is the highest priority device. Interrupt priorities are established by the ICU, and the code in `loadIdt()` chooses the default priority scheme (IRQ0 highest, IRQ14 lowest). It can be changed.

3.6.2 The PIT: i8253

See the hardware section for more information on this chip.

The sample kernel implementation uses this chip configured for interrupts every 20^{th} of a second. This is a useful rate, but limits delay resolution accordingly.

3.6.3 The RTC: DS12887

See the hardware section for more information on this chip.

The interrupt generated by this chip can be used for time slices, and it can be configured to one of 13 different rates. The interrupt rates are shown in Table 3.1. The documentation for the chip says it is accurate to "±1 minute per month at 25°C". What this means for the accuracy of the periodic interrupts is not clear.

3.7 Graphics tips

Please see the `graphics(3)`, `libsvga(3)` and `libvgagl(3)` pages for detailed information on using graphics.

These libraries work by mapping the frame buffer into the processes address space and then writing directly to the frame buffer. Somehow, they must map the video card into their address space. To support this, you must provide an implementation of the `MemMap()` kernel call. The function `MemMap()` maps a physical address into a pointer in the data segment of calling process. If you have any memory protection it must be "loosened" for the process that calls `MemMap()`.

You definitely want to localize all graphics to one executable so that you do not spend time downloading multiple copies of `libvgagl.a` and `libvga.a`, both of which are quite large. The graphics libraries allocate memory using `malloc()` and will require a large stack. I have had success with values like 128k as a stacksize.

I recommend using the default video mode (number 19: 640x480x24b colour). There are higher-resolution video modes, but they do not have as many colours. Using true-colour obviates the need to allocate colours from a palette.

3.8 Sound tips

The EOS systems have a Sound Blaster 16 sound card installed. A library has been provided for accessing some features of this card, but it is extremely limited. Read `sound(3)` for more information on this library.

After the library was written, a useful file was discovered on the net. This file documents most of the features of the card in great detail (see `sound(3)` for the file's location). You will probably want to take the information in this file and construct your own library for doing sound. However, in order to use the DAC (Digital to Analog Converter) to play back samples, you will have to master DMA. This might be considered difficult. You may want to spend your time on things that will get you marks first.

3.9 Mixing assembly language and C

You have two choices when mixing assembly language and C. Small amounts of code can be put "inline" while larger amounts can be put into separate files. `i586-elf-gcc(1)` will compile files ending in `.S` using the assembler, `i586-elf-as(1)` after preprocessing them with `i586-elf-cpp(1)`.

To do inline assembly, use the keyword "asm". Here is an example of its usage.

```
static __inline u_int
myDS(void)
{
    u_int val;
    __asm __volatile("pushl %%ds ; popl %0" : "=r" (val));
    return val & 0x0ffff;
}
```

You should read the `gcc(1)` manual to understand the syntax of the `asm` keyword. The syntax is a touch involved because you have to integrate your code with the optimized code `gcc(1)` generates. You can use it trivially in many cases, where you don't need to pass any information between 'C' and assembly:

```
void doNotCallMe()
{
    asm("
        .text
        .global _entryPoint
    _entryPoint:
        pushal
        pushl   %es
        pushl   %ds

        # insert code here to save SS and ESP

        movl    $0x038, %eax
        movl    %eax, %ds
        movl    %eax, %ss
        movl    $(_tempStack+tempStackSize), %esp

        call    _cFunction

        # insert code here to restore SS and ESP
```

```
            popl    %ds
            popl    %es
            popal

            iretl
        ");
    }
```

For an example of assembler syntax, you can look at the output of `gcc(1)`, using the '-S' flag. All symbols declared in C will be preceded by an underscore.

You should check the final asm output of C source files with inline assembly, since `gcc(1)` may not always do what you intended. You may also use `objdump(1)` to disassemble object files.

If you're using C++, then you will almost certainly get linking errors when trying to link your assembly code in with the rest of your system. This is due to the way C++ names linkable entities. If this is the case you must add the phrase 'extern "C" {...}' around either the relevant include statement or each function declaration. This tells the C++ compiler to look for "old style" function names and will allow it to properly link with your assembly code:

```
extern "C" {
#include "abscopy.h"
}

extern "C" {
void absRead(uint from, void *to, int count);
}
```

Index