Chapter 8. Interrupt and Exception System

This chapter contains information on the low-level handling of interrupts. Interrupts can be from internal or external sources. See the chapter on the "159 PC" for details of devices that can generate external interrupts (i.e., interrupt requests).

This chapter discusses the handling of interrupts on an X86 processor. Some interrupts support handling of "exceptional conditions" that have been caused by the instruction stream. Other handle attention requests from hardware devices. When this chapter discusses the actions taken by the CPU, it is assumed to be running in 32-bit protected mode.

Intel separates interrupts into two broad categories: *exceptions*, generated internally and are synchronized with, or caused by, the instruction being executed, or *interrupts* from external devices running asynchronously with to program. Interrupts are requested change of control and/or privilege levels. Exceptions are abnormal error handling that can occur even midway during the execution of an instruction. When an interrupt occurs, special handlers are invoked. Divide by 0 is a synchronous exception because given the same inputs this exception will always occur on the divide instruction. The interrupt from, say a parallel port, arrives whenever the (external) printer indicates it is ready. i.e. asynchronous to the CPU's clock and instruction stream.

Exceptions are handled in three different ways, either as a *trap*, a *fault*, or an *abort*. Generally if the exception is known at an instruction boundary, it is a trap. All software interrupts, including breakpoint, overflow test, and bounds check are traps. After the system handles the trap, execution usually resumes at the next instruction. Faults occur during the processing of an instruction and are not immediately obvious. A page fault occurs because the page just so happens to not be mapped, or a divide by zero occurs. After the system recovers from a fault, the faulting instruction should be retried (i.e. executed again). An abort exception is fatal, and cannot be recovered from. A double exception occurs if the CPU faults while it is moving to an environment to handle a fault, such as trying to push `CS`, `IP` and `EFlags` onto the kernel stack, but it hasn't been mapped, for instance.

Interrupt Vectors (General)

Interrupt handling goes through an interrupt vector numbered 0 to 255. The table below describes the CPU's interrupt vector usage. Note that IRQ vectors are a subset of the processor interrupts. An interrupt is just a signal; it cannot convey any information other than "it happened."

The first 32 interrupt vectors are now reserved by Intel for the CPU. If you compare the name of the interrupt with its usage in an IBM PC, you'll note they don't match. For instance, interrupt 5 is really for bounds checking on the Pentium, but the original IBM PC BIOS used it for doing a print screen. This usage of reserved interrupt vectors is unfortunate. However, since the IBM PC ROM BIOS isn't used in 159, you should ignore the BIOS usage of the lower 32 interrupt vectors.

♦ Software Exceptions

Software can cause any of the 256 interrupts to occur using the two-byte "`INT nn`" instruction. Linux uses interrupt 0x80 for user programs to trap into its kernel. If the calling code isn't in ring-0, the CPU first switches to ring-0 and pushes the current (non-ring-0) `SS:ESP`. It then pushes the flags and `CS:EIP` registers. If trapping via an interrupt gate, interrupts are disabled.

Exception Processing

When the CPU recognizes a protection violation, abnormal result value, say from divide by 0, or illegal instruction requests in the code stream, it generates an exception. This interrupt comes from within the processor. There are also three special one-byte instructions that can cause an exception. These include **INT3** (breakpoint), **INTO** (trap on overflow), and **BOUND** (trap on out-of-bounds).

Exception processing suspends the instruction execution immediately. Some exceptions can be restarted and so the address of the offending instruction is saved. Exceptions cannot be disabled. A trap saves the instruction pointer after the causing instruction because it must not be rerun. For a catastrophic error, the IP value isn't predictable.

There are three classes of exceptions: traps, faults and aborts. A trap is issued after an instruction, e.g. single stepping and breakpoints, and the saved **EIP** is after the trap instruction. A fault occurs during the processing of instruction, where the register state is midway through and **EIP** points to the start of the instruction so after recovery it can be restarted. Aborts are for serious failures, such as invalid system tables or hardware troubles. Since aborts aren't meant to be retried (i.e. result of instruction won't be stored), the **EIP** value is unpredictable.

♦ General Handling

When the Pentium has decided to handle the exception, processing is very similar between interrupts and exceptions. All interrupts and exceptions boil down to an interrupt number and indirection via the interrupt descriptor table into some recovery and handling code. First, the proper interrupt vector number must be obtained. If internal, then the CPU logic generates this number. If external, the CPU asks the interrupt controller for the vector number.

Now the CPU can fetch the correct descriptor within the interrupt descriptor table (IDT), pointed to by the IDTR. The IDT is a table separate from the global descriptor table (GDT) though both have the same format: an array of descriptors. Each eight byte descriptor has a type, and the only allowed types in the IDT are interrupt gate, trap gate and task gate. The task gate causes the CPU to switch its notion of "task" upon interrupt. The difference between a trap gate and an interrupt gate is the latter disables interrupts. The descriptor must be "present" and have a compatible DPL.

The CPU might change to the protection ring number in the descriptor's selector (i.e. given by the lower two bits), called the requested privilege level (RPL). If the current CPL (lower two bits of CS) match the RPL, then the **CS**, **EIP** and **EFlags** can be pushed on the current stack (**SS:ESP**) and no protection level change needs to occur. An interrupt gate will cause interrupts to be disabled. If the CPL is greater than the RPL, then the CPU is transitioning to a greater privilege level, and a ring change must occur.

When a protection ring change occurs, the CPU will consult fields in the task-state segment (TSS). The goal is to move from the current stack to the RPL stack. In the case where the current stack is not mapped and a fault has occurred, for instance, the CPU cannot use the current ring's stack. Memory there just isn't valid. However, it must save the current **SS:ESP** somewhere. The CPU is depending on the fault handler having valid stack memory. If not, the CPU will double fault, A Bad Thing™. These values are needed for when the handler returns. The CPU will load the **SS:ESP** for the RPL ring from the TSS and then push the previous **SS:ESP** values onto it. This leaves a pointer back to the previous environment where the exception occurred. Then normal interrupt processor occurs with the pushing the **CS**, **EIP** and **EFlags**.

Eventually service for the interrupt will be complete. If the interrupt was external (an IRQ request) or software generated (an **INT** instruction), after the **IRET**, the CPU resumes execution at the next machine instruction. If the interrupt was a fault or an abort, like a page fault or general protection fault, the **EIP** points to the instruction that caused the trouble. In this case, the **IRET** will re-try the instruction.

♦ Pentium Exceptions

Exceptions include the events detailed below. Vector 2 is the non-maskable interrupt.

- Division by 0 (interrupt vector 0): When the integer divisor is 0, this fault occurs.
- Single-Step (interrupt vector 1): When the trap flag is set, after an instruction completes, this exception occurs. Software should check debug register **DR6** to see if it's a fault or a trap.
- Non-masked Interrupt (interrupt vector 2): Occurs when an external device asserts the **/NMI** line. On the IBM PC, it is hooked to the memory parity checking circuit, and to the expansion bus. The real-time clock can disable this signal.
- Breakpoint (interrupt vector 3): The single-byte breakpoint instruction **INT3** causes a breakpoint trap. After handling this trap, the debugger should replace the instruction byte where the **INT3** was stored and retry the instruction.
- Overflow test, **INTO** (interrupt vector 4): When the overflow flag is set and an **INTO** instruction executes, the processor generates this trap.
- **BOUND** check (interrupt vector 5): If an out-of-range index is checked with a **BOUND** instruction, this fault is taken.
- Invalid opcode (interrupt vector 6): To provide for future opcode additions, unknown and invalid opcodes cause this fault. This can be caused by an aberrant jump into data, or incorrect use of the **LOCK** prefix. Good instructions with illegal operands generate invalid opcode faults.
- No Coprocessor Available (interrupt vector 7): When the coprocessor is requested but the **CR3** flag bit states the coprocessor isn't available (either not owned or hardware not installed), this fault occurs.
- Double Fault (interrupt vector 8): Generated when the processor is currently handling one exception and another error occurs (like virtual memory page holding IDT isn't present). Sometimes these errors can be handled one-at-a-time, but if not this abort occurs. This is generally a catastrophic condition.
- Stack exception (interrupt vector 12): After so many pops, the **ESP** offset wants to wrap around to 0, then this fault occurs.
- General Protection Fault (interrupt vector 13): When the offset specifies a location not allowed by the segment descriptor, this fault occurs.[1] For stack segments, this means a value *below* the limit. A GPF is related to segments and not paging. All general protection faults are restartable.
- Page Fault (interrupt vector 14): An address from an instruction tried to access memory in an invalid manner. This could be because of insufficient privilege, or page frame or page table not present. This fault pushes an error code.
- Coprocessor Error (interrupt vector 16): Any coprocessor can generate exceptions that the main CPU causes a fault to occur.
- Alignment Check Exception (interrupt vector 17): Normally the i386 processor is oblivious to alignment of multi-byte values. Usually fetching a double word from an odd address is no cause for alarm. However, when the following conditions are true, alignment checking is performed:
    - The AM flag in **CR0** is set;
    - The AC flag in **EFLAGS** is set; and
    - With protected mode, the CPL is 3 (user mode, least privileged).
- Machine Check (interrupt vector 18): When the processor detects an internal error condition, it aborts. This can be caused by internal parity errors.

---

[1] MS-Windows users are always asking the question, "Who is General Protection, why is it his fault, and what is he doing reading my computer's memory?"

See “Hardware Interrupts” for how **IRQ** interrupt vectors are used.

Event #	Description	Source	Treatment
0	Divide by Zero	**DIV** and **IDIV** instruction	Fault
1	Single-step enabled	Any code of data reference	Trap/Fault
2	Non-Maskable Interrupt	Hardware input	Trap
3	Breakpoint	**INT3** instruction	Trap
4	Overflow tested positive	**INTO** instruction	Trap
5	BOUND check	**BOUND** instruction	Fault
6	Invalid Opcode	Reserved opcodes	Fault
7	No Coprocessor Available	**ESC** and **WAIT** instructions	Fault
8	Double Fault (A Bad Thing™)	Any instruction	Abort
9	Coprocessor segment overrun	**ESC** instruction	Abort
10	Switch to Invalid TSS	**JMP**, **CALL** or **IRET** instructions	Fault
11	Segment Not Present	Any instruction that changes segments	Fault
12	Stack Exception	Stack operations	Fault
13	General Protection Fault	Any code or data reference	Trap/Fault
√ 14	Page not present	Any code or data reference	Fault
15	Reserved		
16	Co-processor exception	**ESC** and **WAIT** instructions	Fault
17	Alignment Check (when enabled)	Any code or data reference	Fault
18 (19-31 unused)	Machine Check from bus check or external parity error	Internal or external hardware	Abort
√32-47	External Interrupt Request	Reprogrammed for IRQ’s for SPEDE	External
√48-255	Software Interrupts (**INT nn** instruction)	**INT nn** instruction	Trap

Interrupt Service Requests (IRQs)

External interrupt request lines are mapped to a range of CPU interrupt vectors. In the original IBM PC/AT architecture, the first eight IRQs map to one range, and the second eight map to another range. To simplify the handling of external interrupt requests, the SPEDE runtime sets them to sixteen consecutive interrupt vectors. These vectors are 0x20 to 0x2F, which map IRQ 0 to IRQ 15. Thus, interrupt vector 0x2***N*** services IRQ-***N***.

The following table describes what device is connected to each interrupt request line. IRQs 9 and 11 are available for adapter boards (i.e. don’t have system-reserved uses). IRQ 8 is from the real-time clock alarm; this could be used for times of many seconds (or days), permitting the CPU to not run while waiting. IRQ 10 is commonly used by network adapters. Note some IRQs have two or more sources: the interrupt service routines (ISRs) need to check all possible sources. IRQs with multiple sources use a level triggered interrupt. The Intel 82371FB PCI-to-ISA bridge chip (used in many Pentium-based systems) can reconfigure many of these interrupt signals.

IRQ	Interrupt Vector	Priority Ranking	Usage
✓ IRQ 0	0x20	1	Channel 0 from i8253 periodic timer
IRQ 1	0x21	2	Keyboard
IRQ 2	**0x22**	**3**	*Cascade for IRQ 8 to IRQ 15.*
✓ IRQ 3	0x23	12	Serial ports #2, #4, #6 and #8
✓ IRQ 4	0x24	13	Serial ports #1, #3, #5 and #7
IRQ 5	0x25	14	Parallel port #2
IRQ 6	0x26	15	Floppy disk drive
✗ IRQ 7	0x27	16	Parallel port #1
IRQ 8	**0x28**	**4**	Real-time clock (alarm)
IRQ 9	0x29	5	VGA timing interrupt (vertical retrace)
IRQ 10	0x2A	6	*Unused*
IRQ 11	0x2B	7	*Unused*
IRQ 12	0x2C	8	PS/2-style mouse
IRQ 13	0x2D	9	Numeric co-processor. Use interrupt 16 instead
IRQ 14	0x2E	10	IDE hard disk primary controller
IRQ 15	0x2F	11	IDE secondary controller (if installed and enabled)
IRQ 16	—	17	Inter-Processor Interrupt (from APIC)

Table 8-2. IRQ Descriptions

IRQ 2 is a cascade for IRQ 8 to IRQ 15. In the beginning on the IBM PC, there were only eight interrupt sources. However, when the hard disk option and math coprocessor option were added in the PC/AT, more interrupts were required. A second interrupt controller (slave) was added, and its output goes to IRQ 2 of the first (master). This perverts the interrupt priority ranking because IRQ 15 has more priority than IRQ 3! To show the true priority ranking, it would be better to name IRQ 8 to IRQ 5 as IRQ 2.1 to IRQ 2.8 instead. However, when it comes to interrupt vectors, they are as listed in the table above.

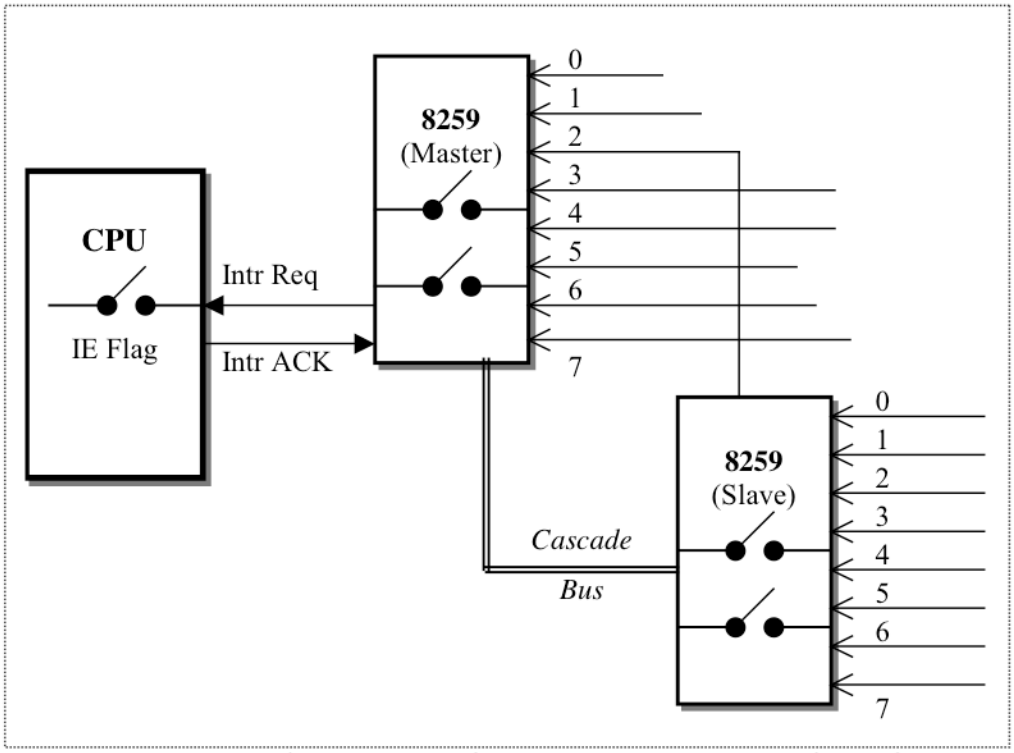
IRQ 16 is used in multi-processor systems. The Advanced PIC, internal to the Pentium II (and late model Pentiums) generates this request. The interrupt controller for IRQ 0 to IRQ 15 is now implemented as part of the PCI support chip-set.

♦ Hardware Interrupts

External hardware interrupts are first processed by the Programmable Interrupt Controller (PIC) chips. This logic (contained in two i8259 chips in the IBM PC/AT) are told which interrupts are allowed to pass thru and interrupt the CPU and which are currently held off (i.e. masked). When they are allowed to reach the CPU, the hardware interrupt can be held off if interrupts are disabled via the flags register (**IF** flag, bit 9).

When an interrupt request is generated, the 8259A decides if it should notify the CPU. If a higher priority IRQ is being serviced, the new request is put on hold. If the news priority is higher, or no IRQ is being serviced, the master 8259A will signal the CPU. When the CPU acknowledges it, the PIC which has the highest priority IRQ will output its local IRQ number (0 through 7), plus an offset value in the high five bits. The CPU reads this number as an interrupt vector and proceeds with interrupt processing.

If the processor has interrupts enabled, then it acknowledges the interrupt request (**/INTA**), and generates a special bus cycle to read the interrupt vector number. Each PIC chip services eight interrupts, and is programmed with the upper five bits of the interrupt vector (the lower 3 bits relate to



which interrupt line was asserted). In the IBM PC standard architecture, the first PIC directs to vectors 0x08 to 0x0F, and the second to 0x70 to 0x77. For 159, IRQ 0 to IRQ 15 are mapped to vectors 0x20 to 0x2F. This way they're contiguous, and don't interfere with the i386 processor-reserved interrupt vectors 0x00 to 0x01F. Therefore, which interrupt vectors service what hardware is left up to system designers. It isn't dictated by the processor architecture.

Once software has serviced the source of the interrupt, it must send an acknowledge command back to the 8259A chip. It can either be an automatic or specific End-of-Interrupt (EOI) command. The automatic EOI is sent to all 8259A chips; the specific goes first to the slave that accepted the interrupt request and then the master 8259.

♦ Using the 8259A Programmable Interrupt Controller

Each 8259A must be initialized when the operating system boots up. Fortunately, this initialization is done for you by the startup code linked into your program. Each chip handles eight IRQ lines. IRQ 2 of the master is fed by the slave 8259A whenever it has any IRQ request. Each 8259A occupies two I/O byte ports: the master starts at 0x20 and the slave at 0xA0. After each interrupt is serviced, the software must acknowledge the interrupt. For the IBM PC architecture, the inputs are edge triggered.

To initialize, you would output an initialization control word to the 8259A base address (ICW1), then three bytes to I/O port base+1. Lastly, two operation control words are sent to each chip to mask interrupts and preset an operation mode.

If a device driver wants to handle the interrupt, it must be unmasked in the appropriate 8259A chip. (If the CPU has interrupts enabled, the IRQ will cause an interrupt in the CPU.) To allow an interrupt, the correct bit in the IMR register must be cleared to 0. If IRQ 8 to IRQ 15 is being unmasked, then IRQ2 must also be unmasked. Otherwise, the master won't respond to requests from the slave.

Once the interrupt controller tells the CPU about a request, the CPU must manually acknowledge it. You do this by writing a command to dismiss the particular interrupt request. Use the macro `SPECIFIC_EOI()` from <*spede/machine/pic.h*> for this. (If the IRQ is on the slave, you must clear IRQ 2 first.) This code sample works for the parallel port.

```
1     #include <spede/machine/pic.h>            /* For PIC defines */
2     #include <spede/machine/io.h>             /* For outportb() */
3     #include <spede/assert.h>
4
5     void irq_parallel_enable(void)
6     {
7         /* Read current settings, then clear bit 7 to 0 (enable) */
8         outportb(0x21, (~BIT(IRQ_PAR1)) & inportb(0x21));    /* Unmask IRQ 7 */
9     }   /* end irq_parallel_enable() */
10
11    void irq_ack_parallel(void)
12    {
13        outportb(0x20, SPECIFIC_EOI(IRQ_PAR1));              /* Ack IRQ 7 */
14    }   /* end irq_ack_parallel() */
```

Port 0x21 is the IMR of the master PIC. It tells which interrupt requests are masked. A one bit means the request input is masked and thus is ignored. This register is both read and write capable. Port 0x20 is the master PIC command port. (I/O port 0xA1 is the slave's IMR, and port 0xA0 is the slaves command register.) Once a request has been cleared, the device can issue subsequent requests. Without this (manual) acknowledging cycle, the device can never re-trigger the PIC.

Note the above code is not very general. Below is a function to acknowledge any one of the 16 interrupt requests. The OS must also clear the interrupt request logic of the particular device (usually done by a device driver). Note "IRQ_CASCADE" is from the <*spede/machine/pic.h*> header file.

```
15    void irq_ack(unsigned irq)
16    {
17        assert( irq < NR_IRQS );
18        if( irq >= 8 ) {
19                  outportb(0x20, SPECIFIC_EOI(IRQ_CASCADE));
20                  outportb(0xA0, SPECIFIC_EOI(irq - 8));
21        } else {
22                  outportb(0x20, SPECIFIC_EOI(irq));
23        }
24    }   /* end irq_ack() */
```

Here is a generalized routine to enable a single interrupt request. IRQ 3 is asserted when any interrupt from the second interrupt controller is asserted. Your code must install a software handler for the IRQ before enabling it.

```
25    void irq_enable(unsigned irq)
26    {
27        assert( irq < NR_IRQS );
28        /*
29         * Unmask it in 8259 PIC. Read current mask, turn off bit to enable,
30         * then store new mask back to PIC.
31         */
32        if( irq >= 8 ) {
33            outportb( 0x21, inportb(0x21) & ~BIT(IRQ_CASCADE) );
34            outportb( 0xA1, inportb(0xA1) & ~BIT(irq-8) );
35        } else {
36            outportb( 0x21, inportb(0x21) & ~BIT(irq) );
37        }
38    }   /* end irq_enable() */
```

Appendix C: Chip Dictionary

Often times in this documentation, a support chip is referred by to its number rather than its function. Since the original IBM PC was built from readily available parts, this practice is common. This appendix lists in numeric order important chips in the IBM PC/AT machine. Since this table is numbers, the Pentium CPU is listed as 80586. Note that numeric order isn't the same as when the chip was first produced. For example, the MC146818 was used as part of the 80286 computer.

Chip Number	Name	Description
µPD765	Floppy Disk	Original floppy disk controller used in IBM PC.
6845	Video	Original video controller used in basic video adapter boards.
8042	CPU	Microcontroller used in IBM PC keyboards.
√ 8088	CPU	Original processor used in IBM PC and XT, with only 20 address lines.
8087	math coproc	Original 8-bit FPU co-processor.
8237A	DMA	Original 16-bit address DMA. PC/AT has two.
8250A	UART	Original UART, superceded by NS16550A.
√ 8253A	PIT	Original Programmable Interval Timer, chip contains 3 timer units.
8254A	PIT	Improved PIT (over i8253)
8255A	PPI	Programmable Peripheral Interface chip with 24 I/O lines.
√ 8259A	PIC	Programmable Interrupt controller, handlers IRQ 0 to IRQ 15
8284	Clock	Generates various clock signals for original 8088. Integrated inside 80386 and above.
8288	Bus Controller	Controls the 8088 system bus. Integrated inside 80386 CPUs and above.
NS16540	UART	Fast UART from National Semiconductor
√ NS16550A	UART	Faster UART with 16-byte receive and transmit FIFO's.
NS16650	UART	Even faster UART with 32 byte transmit and receive FIFO's.
80286	CPU	16-bit CPU with 24 address lines.
80287	math coproc	Floating-point unit with 16-bit data bus.
82072	Floppy disk	Floppy disk controller used in 80286 (and up) machines.
82284	Clock	Generates various clock signals for 80286.
82288	Bus Controller	Controls the 80286 system bus.
82450	UART	Intel's successor to the 8250.

Chip Number	Name	Description
80386	CPU	32-bit CPU with 32 address lines.
80387	math coproc	Faster running floating-point unit with a 32-bit data bus.
82360SL	Support	High integration support chip. It contains power management, real-time clock, CMOS non-volatile RAM, two UART's, a 15 IRQ interrupt controller, 6 channel DMA controller, and 6 timer units. Also contains keyboard, hard disk and floppy disk interface logic, all wired for an IBM PC architecture.
80486	CPU	32-bit CPU with 32 address lines, on-board floating-point unit and memory cache.
80586 (Pentium)	CPU	32-bit CPU with 36 address lines, pipelining, and on-board L1 cache.
82371	PCI Bridge	PCI-to-ISA Bridge plus various support "chips," including timers, DMA and interrupt controllers (like 82360SL but includes PCI bridge logic).
82443BX	PCI Host Bridge / Controll	Interfaces with Host Front-Side bus with PCI, AGP and system memory.
82489	PIC	Outboard Advanced Programmable Interrupt Controller, useful in multi-CPU systems.
80686 (Pentium II)	CPU	32-bit CPU with 36 address lines, pipelining, and on-board L1 and L2 caches.
MC146818	RTC	Battery backed-up real-time clock.

Appendix D: Display Characters

This appendix describes the displayable character set of the IBM PC video adapters, commonly referred to as PC-8. The first 97 displayable characters correspond to the ASCII character set (codes 0x20 to 0x7F). The first 128 codes are the same for ISO-8859 Latin 1.

Symbols in rows `Bx`, `Cx` and `Dx` go to the edge of the character cell and can be used for screen drawing. Note characters 0x00, 0x20 and 0xFF are blanks.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
1x	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2x		!	“	#	$	%	&	‘	(	)	*	+	,	–	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	Δ
8x	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9x	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ő	Ü	¢	£	¥	Pts	₣
Ax	á	í	ó	ú	ñ	Ñ	a	o	¿	⌐	¬	½	¼	¡	«	»
Bx	▒	▒	▓	|	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	┘	┐
Cx	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
Dx	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	▉	▄	▌	▐	▀
Ex	α	β	Γ	π	Σ	σ	µ	τ	Φ	θ	Ω	ζ	∞	Φ	ε	∩
Fx	≡	±	≥	≤	⌠	⌡	÷	≈	O	●	•	√	ŋ	²	■	

This table is more properly called a font table. The font “Times Roman” is used here since its slightly fruity serif accents emphasizes the shape and contour of each glyph. Note, do not depend on every printer rendering all of the PC-8 character set.

Appendix I: I/O Map

This I/O map is ordered by address port (see page 540 in [Messmer, 1997]). The I/O port is in hexadecimal. Typically, PCI and EISA devices are dynamically mapped to I/O ports between $1000 and $BFFF. The original serial chip was the Motorola 8250, although now the National Semiconductor NS16550 is used. See Appendix C, Chip Dictionary, for details about the "Chip Type" column.

I/O address 0x080 is reserved and will never to assigned to a device. It is used by the `IO_DELAY()` macro.

If you go looking for these individual chips in a modern PC, you won't find them. Thanks to high-integration ASIC chips, many functions have been integrated into one IC. For instance, many PCI bridge chips also include a dynamic RAM refresh circuit, thus DMA channel 0 is no longer used for that purpose. A "super I/O" chip might include parallel, serial, floppy disk, DMA, timers and IDE interface into a single 200-pin chip.

I/O Port	Purpose	Bit Size	Chip Type	Additional Notes
000	Address for DMA channel 0			Store low-byte, then high byte
001	Count for DMA channel 0			
002	Address for DMA channel 1			Store low-byte, then high byte
003	Count for DMA channel 1			
004	Address for DMA channel 2			Store low-byte, then high byte
005	Count for DMA channel 2			
006	Address for DMA channel 3			Store low-byte, then high byte
007	Count for DMA channel 3			
008	DMA status/command register	RW 8		
009	DMA Request	W 8		
00A	DMA Channel Mask	W 8		
00B	DMA Mode	W 8		
00C	Reset flip/flop	W 0		Reset by writing to output port
00D	DMA Intermediate Reg	R 8		
00D	Master Clear	W 0		Clear by writing to output port
00E	Clear Mask	W 0		Reset by writing to output port
00F	DMA Mask register	W 8		
√ 020	Init Control Word 1, OCW 2 and 3	RW 8	I8259	Control Register
√ 021	Operation Control Word 1	RW 8	I8259	Mask Register

040	Counter #1	RW 8	PIT	Programmable Interval Timer
041	Counter #2	RW 8	PIT	
042	Counter #3	RW 8	PIT	
043	Control Reg	W 8	PIT	I8253 has read-only control register.
048	Counter #1	RW 8	PIT	
049	Counter #2	RW 8	PIT	
04A	Counter #3	RW 8	PIT	
04B	Control Reg	RW 8	PIT	I8254 allows reading control register.
060	Keyboard scancode/config switches #1		I8255	Set Port B, bit 7 to 0 for keyboard data.
061	Port B: system control	RW 8	I8255	Bit 7 = 0 for keyboard
062	Port C: <1 Meg RAM settings		I8255	Usage deprecated
063	8255 mode control	RW 8	I8255	
064	Keyboard and mouse Status/Control port	RW 8		
070	RTC address	RW 8	MC146818	Use read-modify-write to select a data byte. Bit 7 controls NMI.
071	RTC Data	RW 8	MC146818	
074	Extended CMOS low-order address	W 8		$075,$074 used on PS/2 system. Write low-order address byte first. Then read or write port $076.
075	Extended CMOS high-order address	W 8		
076	Extended CMOS data port	RW 8		
080	**Reserved**			**Will never be assigned.** The `IO_DELAY()` macro uses this port.
081	DMA Page Register 2			
082	DMA Page Register 3			
083	DMA Page Register 1			
084	**Reserved**			**Will never be assigned.**
085	???			
086	???			
087	DMA Page Register 0			
088	???			
089	DMA Page Register 6			
08A	DMA Page Register 7			
08B	DMA Page Register 5			
08F	DMA Page Register 4			DRAM Refresh
090	Microchannel Bus Arbitration	RW 8		
094	MCA motherboard activation	RW8		Select motherboard available through POS registers.

096	MCA Adapter activation	RW 8		Selects which adapter is available through POS registers.
0A0	Iniitialize control word #1, OCW2 and 3		I8259	IRQ 8 – 15 (slave unit)
0A1	ICW 2,3,4, and OCW 1		I8259	
0C0	Address for DMA channel 4			Store low-byte, then high byte
0C1	Count for DMA channel 4			
0C2	Address for DMA channel 5			Store low-byte, then high byte
0C3	Count for DMA channel 5			
0C4	Address for DMA channel 6			Store low-byte, then high byte
0C5	Count for DMA channel 6			
0C6	Address for DMA channel 7			Store low-byte, then high byte
0C7	Count for DMA channel 7			
0D0	DMA status/command register	RW 8		
0D2	DMA Request	W 8		
0D4	DMA Channel Mask	W 8		
0D6	DMA Mode	W 8		
0D8	Reset flip/flop	W 0		Reset by writing to output port
0DD	DMA Intermediate Reg	R 8		
0DA	Master Clear	W 0		Clear by writing to output port
0DC	Clear Mask	W 0		Reset by writing to output port
0DF	DMA Mask register	W 8		
0F0-0FF	Control FPU		80387	
100-107	MCA Programmable Option Selects (POS)	RW 8		Available on each adapter board. Port 096 selects which on.
1F0	Data register	RW 16	IDE	
1F0	Error register	R 8	IDE	
1F1	Pre-compensation	W 8	IDE	Value ignored nowadays
1F2	Sector Count	RW 8	IDE	
1F3	Sector Number	RW 8	IDE	
1F4	Cylinder – LSB	RW 8	IDE	10 bits used for IDE
1F5	Cylinder – MSB	RW 8	IDE	16 bits used for EIDE
1F6	Drive/head select	RW 8	IDE	
1F7	Status register	R 8	IDE	
1F7	Command register	W 8	IDE	

201	Game adapter status	R 8		Upper 4 bits are trigger buttons.
274-277	PnP Control ports	RW 8		Plug-and-Play controller/enumerator.
278	LPT2 data port	RW 8		Output 0 before reading.
279	LPT2 status port	RW 8		
27A	LPT2 control port	RW 8		
2E8	COM4 receive/transmit buffer	RW 8	I8250	IRQ 3
2E9	COM4 interrupt enable	RW 8	I8250	
2EA	COM4 interrupt identification	RW 8	I8250	
2EB	COM4 Data format		I8250	Line control register
2EC	COM4 Modem control	R	I8250	RS-232 outputs
2ED	COM4 Serialization status	RW 8	I8250	Line status register
2EE	COM4 Modem status	W 8	I8250	RS-232 input
2EF	COM4 scratch pad	RW 8	16550	Not used by chip
2F0	COM5 receive/transmit buffer	RW 8	I8250	IRQ 4
2F1	COM5 interrupt enable	RW 8	I8250	
2F2	COM5 interrupt identification	RW 8	I8250	
2F3	COM5 Data format		I8250	Line control register
2F4	COM5 Modem control	R	I8250	RS-232 outputs
2F5	COM5 Serialization status	RW 8	I8250	Line status register
2F6	COM5 Modem status	W 8	16550	RS-232 input
2F7	COM5 scratch pad	RW 8	I8250	Not used by chip
2F8	COM2 receive/transmit buffer	RW 8	I8250	IRQ 3
2F9	COM2 interrupt enable	RW 8	I8250	
2FA	COM2 interrupt identification	RW 8	I8250	
2FB	COM2 Data format		I8250	Line control register
2FC	COM2 Modem control	R	I8250	RS-232 outputs
2FD	COM2 Serialization status	RW 8	I8250	Line status register
2FE	COM2 Modem status	W 8	I8250	RS-232 input
2FF	COM2 scratch pad	RW 8	16550	Not used by chip
378	LPT1 data port	RW 8		Output 0 before reading.
379	LPT1 status port	RW 8		
37A	LPT1 control port	RW 8		
3BC	LPT0 data port	RW 8		Output 0 before reading. LPT0 on monochrome monitor adapter, which has been discontinued.
3BD	LPT0 status port	RW 8		
3BE	LPT0 control port	RW 8		

3E0	**PCMCIA index port**	**W 8**		
3E1	**PCMCIA data port**	**RW 8**		
3E0	**COM6 receive/transmit buffer**	**RW 8**	**I8250**	
3E1	**COM6 interrupt enable**	**RW 8**	**I8250**	
3E2	COM6 interrupt identification	RW 8	I8250	
3E3	COM6 Data format	R 8	I8250	Line control register
3E4	COM6 Modem control	R 8	I8250	RS-232 outputs
3E5	COM6 Serialization status	RW 8	I8250	Line status register
3E6	COM6 Modem status	W 8	I8250	RS-232 input
3E7	COM6 scratch pad	RW 8	16550	not used by chip
√ 3E8	COM3 receive/transmit buffer	RW 8	I8250	IRQ 4
√ 3E9	COM3 interrupt enable	RW 8	I8250	
√ 3EA	COM3 interrupt identification	RW 8	I8250	
√ 3EB	COM3 Data format		I8250	Line control register
√ 3EC	COM3 Modem control	R	I8250	RS-232 outputs
3ED	COM3 Serialization status	RW 8	I8250	Line status register
√ 3EE	COM3 Modem status	W 8	I8250	RS-232 input
3EF	COM3 scratch pad	RW 8	16550	Not used by chip
3F1	Status reg A and B	R 8		Floppy disk
3F2	Digital Output Register (DOR)	RW 8		
3F4	Main Status/	R		
3F4	Data rate select	W		
3F5	Data Register	RW 8		Send commands and receive status here.
3F7	Configuration control	W		
3F8	COM1 receive/transmit buffer	RW 8	I8250	IRQ 4
3F9	COM1 interrupt enable	RW 8	I8250	
3FA	COM1 interrupt identification	RW 8	I8250	
3FB	COM1 Data format		I8250	Line control register
3FC	COM1 Modem control	R	I8250	RS-232 outputs
3FD	COM1 Serialization status	RW 8	I8250	Line status register
3FE	COM1 Modem status	W 8	I8250	RS-232 input
3FF	COM1 scratch pad	RW 8	16550	Not used by chip
462	Software NMI trigger	W	EISA	Any write causes NMI
48B	???	RW 16		????
4D0	Edge/Level triggering of IRQ7..IRQ3	RW 8	82371	Bits <7:3> 0=edge, 1=level trigger for IRQs. Bits <2:0> always 0.

4D1	Edge/Level triggering of IRQ15, IRQ 14, IRQ12..IRQ9	RW 8	82371	Bits <7:6>, <4:1> 0=edge, 1=level trigger for IRQs. Bits <5> and <0> always 0.
0800-08FF	EISA CMOS non-volatile RAM	RW		Extended-ISA/Micro-channel. Page selected by write to $C00.
0C00	EISA CMOS page	W		Controls data page for $800-8FF (Non-volatile CMOS RAM).
0CF8	PCI address pointer	RW 32	PCI	
0CFC	PCI data register	RW 32	PCI	
C000-CFFF	PCI configuration area	RW 32	PCI	Used in configuration method #2, which has been deprecated.

We are all agreed that your theory is crazy.
The question which divides us is whether it is crazy enough to have a chance of being correct. My own feeling is that it is not crazy enough.

— NIELS BOHR.