

Overview of Input/Output Mechanisms

Norman Matloff
University of California, Davis
©2001-2007, N. Matloff

February 24, 2007

Contents

1	Introduction	3
2	I/O Ports and Device Structure	3
3	Program Access to I/O Ports	3
3.1	I/O Address Space Approach	4
3.2	Memory-Mapped I/O Approach	4
4	Wait-Loop I/O	5
5	PC Keyboards	6
6	Interrupt-Driven I/O	6
6.1	Telephone Analogy	6
6.2	What Happens When an Interrupt Occurs?	7
6.3	Alternative Designs	8
6.4	Glimpse of an ISR	8
6.5	I/O Protection	9
6.6	Distinguishing Among Devices	9
6.6.1	How Does the CPU Know Which I/O Device Requested the Interrupt?	9
6.6.2	How Does the CPU Know Where the ISR Is?	10

CONTENTS

CONTENTS

6.6.3	Revised Interrupt Sequence	10
6.7	How Do PCs Prioritize Interrupts from Different Devices?	10
7	Direct Memory Access (DMA)	11
8	Disk Structure	12
9	USB Devices	12

1 Introduction

During the early days of computers, input/output (I/O) was of very little importance. Instead computers, as their name implied, were used to **compute**. A typical application might involve huge mathematical calculations needed for some problem in physics. The I/O done in such an application would be quite minimal, with the machine grinding away for hours at a time with little or no I/O being done.

Today, I/O plays a key role in computer applications. In fact, applications in which I/O plays an important role can arguably be viewed as the most powerful “driving force” behind the revolutionary rise to prominence of computers in the last decade or two. Today we make use of computers in numerous aspects of our business and personal lives—and in the vast majority of such uses, the programs center around I/O operations.

In some of these applications, the use of a computer is fairly obvious: First and foremost, the Internet, but also credit-card billing, airline reservations, word processors, spreadsheets, automatic teller machines, real estate databases, drafting programs for architects, video games, and so on. Possibly less obvious is the existence of computers in automobiles, and in such household items as washing machines and autofocus cameras. However, whether the existence of computers in these applications is obvious or hidden, **the common theme behind them is that they all are running programs in which I/O plays an important role, usually the dominant role.**

To give some concrete of the central role played by I/O, consider the example of the ATM machine. Here are some of the I/O devices the computer in the machine will have:

- The motor used to drag the card in and push it out.
- The scanner which reads the magnetic strip on the card.
- The motors used to prepare and push out the cash.
- The screen and keypad.
- The connection to the network (which leads to a central computer for the bank chain).

2 I/O Ports and Device Structure

An I/O device connects to the system bus through **ports**. I/O ports are similar in structure to CPU registers. However, a port is not in the CPU, being instead located between the I/O device and the system bus, in the **interface card** for the device.

Ports are usually 8 bits in width. They have addresses, just like words in memory do, and these addresses are given meaning through the address bus, just as is the case for words in memory.

3 Program Access to I/O Ports

Note that a potential ambiguity arises. Suppose we wish to communicate with port 50, which is connected to some I/O device. The CPU will place the value 50 in the MAR to go out onto the address bus. Since all

items which are connected to the bus will see what is on the address bus, how can they distinguish this 50, intended for an I/O port, from a value of 50 meant to address 50 of memory? In other words, how can the CPU indicate that it wants to address I/O port 50 rather than memory word 50? There are two fundamental systems for handling this problem, described next.

3.1 I/O Address Space Approach

Recall that the system bus of a computer consists of an address bus, a data bus, and a control bus. The control bus in many computers, such as those based on Intel CPUs, has a special line or lines to indicate that the current value on the address bus is meant to indicate an I/O port, rather than a word in memory. We will assume four such lines, MR (memory read), MW (memory write), IOR (I/O read) and IOW (I/O write).

To access I/O port 50, the CPU will assert either IOR or IOW (depending on whether it wants to read from or write to that port), while to access word 50 in memory, the CPU will assert either MR or MW. In either case the CPU will place the value 50 on the address bus, but the values it simultaneously places on these lines in the control bus will distinguish between I/O port 50 and word 50 of memory.

The programmer himself or herself controls which of these lines will be asserted, by choosing which instruction to use.

```
movb 50, %al
```

and

```
inb $50, %al
```

will both cause the value 50 to go out onto the address bus, and in both cases the response will be sent back via the data bus. But the MOV will assert the MR line in the control bus, while the IN instruction will assert the IOR line in that bus. As a result, the MOV will result in copying the memory byte at address 50 to AL, while the IN will copy the byte from I/O port 50 to AL.¹

Since the lines IOR/IOW and MR/MW allow us to distinguish between I/O ports and memory words having identical addresses, we can describe this by saying that the I/O ports have a separate **address space** from that of memory. Thus we will call this the **I/O address space approach**.

Note the profound hardware dependency here. The asserting of IOR/IOW or MR/MW comes from using IN/OUT or MOV, respectively, and thus the machine must have such instructions.

3.2 Memory-Mapped I/O Approach

Another approach to the “duplicate port/memory address” problem described above is called **memory-mapped I/O**.²

¹You may wonder why the dollar sign is needed in the second instruction, which seems inconsistent with that of the first instruction. But this comes from Intel tradition, so we are stuck with it.

²Unfortunately, many terms in the computer field are “overloaded.” It has become common on personal computers to refer to the mapping of video monitor pixels to memory addresses also as “memory-mapped I/O.” However, this is not the original use of

4 WAIT-LOOP I/O

Under this approach, there are no special lines in the control bus to indicate I/O access versus memory access (though there still must be one or two lines to distinguish a read from a write, of course), and thus no special I/O instructions such as IN and OUT. One simply uses ordinary instructions such as MOV, whether one is accessing I/O ports or memory. Of course, the people who put the computer together must avoid placing memory at the addresses used by I/O ports.

An advantage of memory-mapped I/O is that it can be done directly in C, e.g.:

```
char c,*p;
...
p = 50;
c = *p;
```

Say **c** and **p** were global variables. Then the compiler will likely translate the line

```
c = *p;
```

to something like

```
movl p, %ebx
movb (%ebx), %al
movb %al, c
```

So you can see that from the CPU's point of view, reading that I/O port, which is done by the instructions

```
movb (%ebx), %al
```

is the same as reading from memory. Indeed, the CPU is not aware of the fact that we are reading from an I/O port, which has a very different operation and physical structure from memory.

4 Wait-Loop I/O

Consider the case of reading a character from the keyboard. The program which will read the character has no idea when the user will hit a key on the keyboard. How should the program be written to deal with this time problem? (Note that we are for the moment discussing programs which directly access the keyboard, rather than going through the operating system, as is the case with programs you usually write; more on this point later.)

On an Intel-based PC, the keyboard data port (KDP) has address 0x60, and its status port (KSP) is at 0x62. Among other things, the KSP tells us whether a key has been typed yet; Bit 4 will be 1 if so, 0 otherwise.³

Bit 5 of the KSP plays a role too. When we read the character, we have to notify the keyboard hardware that we've gotten it, so that it can give us another character when one is typed. To do this, we briefly set Bit 5 to 1 and then back to 0. This will cause Bit 4 of the KSP to revert to 0 too.

the term, and should not be confused with what we are discussing here.

³As usual, we are naming the least significant bit "Bit 0," etc.

6 INTERRUPT-DRIVEN I/O

Wait-loop I/O then consists of writing a loop, in which the program keeps testing Bit 4 of the KSP until that bit indicates “character ready,” and then read the character from KDP into some register:⁴

```
1      # loop around until a character is ready
2 lp:
3     inb $0x62, %bl      # get value of KSP
4     andb $0x10, %bl    # check Bit 4
5     jz lp              # if that bit is 0, try again
6 ready:
7     # get the character
8     inb $0x60, %cl
9     # acknowledge getting the character
10    # set Bit 5 of KSP to 1
11    orb $0x20, %bl
12    outb %bl, $0x62
13    # reset Bit 5 of KSP to 0
14    andb $0xdf, %bl
15    outb %bl, $0x62
16 done:
```

During the time before the user hits a key, the value read from the KSP will always have a 0 in Bit 4, so that the AND results in 0, and we jump back to `lp`. But eventually the user will hit a key, resulting in Bit 4 of the KSP being 1, and we leave the loop. We then pick up the character in the code starting at `done`.

5 PC Keyboards

On Intel- (or Intel clone-) based PCs, the keyboard is not directly wired for ASCII. Instead, each of the keys has its own **scan code**—actually two codes, one which the keyboard emits when a key is pressed and the other, 128 more than the first code, which the keyboard emits when the key is released. For example, the A key has codes `0x1e` and `0x9e`, respectively. The keyboard driver will then convert to ASCII.

Say for instance that the user wishes to type ‘A’, i.e. the capital letter. She may hit the left Shift key (code `0x2a`), then hit the A key (code `0x1e`), then release the A key (code `0x9e`) then release the left Shift key (code `0xaa`). The keyboard driver can be written to notice that when the A key is hit, the Shift key has not yet been released, and thus the user must mean a capital ‘A’. The driver then produces the ASCII code `0x41`.

6 Interrupt-Driven I/O

6.1 Telephone Analogy

Wait-loop I/O is very wasteful. Usually the speed of I/O device is very slow compared to CPU speed. This is particularly true for I/O devices which are mainly of a mechanical rather than a purely electronic nature, such as printers and disk drives, and thus are usually orders of magnitude slower than CPU actions. It is even worse for a keyboard: Not only is human typing extremely slow relative to CPU speeds, but also a lot of the time the user is not typing at all; he/she may be thinking, taking a break, or whatever.

⁴By the way we won’t have the character in ASCII form. Instead, we will have it as a **scan code**. More on this below.

Accordingly, if we use wait-loop I/O, the CPU must execute the wait loop many times between successive I/O actions. This is wasted time and wasted CPU cycles.

An analogy is the following. Suppose you are expecting an important phone call at the office. Imagine how wasteful it would be if phones didn't have bells—you would have to repeatedly say, “Hello, hello, hello, ...” into the phone until the call actually came in! This would be a big waste of time. The bell in the phone frees you from this; you can do other things around the office, without paying attention to the phone, because the bell will notify you when a call arrives.

Thus it would be nice to have an analog of a telephone bell in the computer. This does exist, in the form of an **interrupt**. It takes the form of a pulse of current sent to the CPU by an I/O device. This pulse forces the CPU to suspend, i.e. “interrupt,” the currently-running program, say “X,” and switch execution to another procedure, which we term the **interrupt service routine (ISR)** for that I/O device.⁵

The ISR is usually known as the *device driver* for that I/O device. It is typically part of the operating system, but it of course exists on systems without an OS too.

6.2 What Happens When an Interrupt Occurs?

Think for example of the keyboard or any other I/O device which receives characters.

There will be an interrupt request line (IRQ) in the control bus. When an I/O device receives a character, it will assert IRQ.

Recall that we have described the CPU as repeatedly doing Step A, Step B, Step C, Step A, Step B, etc. where Step A is instruction fetch, Step B is instruction decode, and Step C is instruction execution. Well, in addition, the CPU will check the IRQ line after every Step C it does. If it sees IRQ asserted, it will do a Step D, consisting of the following in the Intel case:

```
CPU pushes current EFLAGS value on stack
CPU pushes current CS value on stack (irrelevant in 32-bit protected mode)
CPU pushes current PC value on stack
CPU does PC <-- ISR addresss
```

The next Step A will, as usual, fetch from wherever the PC points, which in this case will be the ISR. The ISR will start executing.

At the end of the ISR there will be an IRET (“interrupt return”) instruction, which “undoes” all of this:

```
CPU pops stack and placed popped value into PC
CPU pops stack and placed popped value into CS
CPU pops stack and placed popped value into EFLAGS
```

Say this occurs when persons X and Y are both using this machine, say X at the console and Y remotely via ssh. X's and Y's programs take turns using the machine (details in our unit on OS). Say during Y's turn,

⁵This is called a **hardware interrupt** or **external interrupt**. Those who have done prior assembly-language programming on PCs should not confuse this with the INT instruction. The latter is almost the same as a procedure CALL instruction, and is used on PCs to implement systems calls, i.e. calls to the operating system.

X types a character. That causes an interrupt, which will be noticed by the CPU when it finishes Step C of whatever instruction in Y's program it had been executing when X hit the key.

From the above description, you can see that the CPU will make a sudden jump to the ISR, so Y's program will stop running. But the hardware saves the current PC and EFLAGS values of Y's program on the stack, so that Y's program can resume later in exactly the same conditions it had at the time it was interrupted. Note carefully that all of that will be done by the hardware, without a single instruction being executed.

The IRET instruction at the end of the ISR is what restores those conditions. Note for instance that it is crucial that Y's EFLAGS value be restored. If for example Y was in the midst of executing the first of the pair of instructions

```
subl %eax, %ebx
jz x
```

execution of the second, which will occur when Y's program resumes, will check whether the Z (Zero) flag in EFLAGS had been set by the SUB instruction. So, EFLAGS must be saved by the CPU when the interrupt occurs, and restored by the IRET.

Make absolutely SURE you understand that the jump to the ISR did NOT occur from a jump or call instruction in Y's program. Y did nothing to cause that jump. However, the jump back to Y did occur because the OS programmer put an IRET at the end of the ISR.

Note too what happens if Y hits a key while X's program is running. That will cause an interrupt to be felt at Y's CPU, which will result in it sending the character through the network, eventually ending up in the Ethernet data port of the machine where X is—causing an interrupt on THAT machine.

6.3 Alternative Designs

By the way, note that when an interrupt occurs, the CPU doesn't even see it until it finishes Step C of whatever instruction was executing at the time the INTR line was asserted. It could be designed differently, by setting up the circuitry so that instructions would be interruptible in the middle. That would reduce interrupt response time a bit, which is good, but at a great expense in complexity. Not only would the circuitry have to save the values of the EFLAGS, CS and PC registers on the stack, but it would also have to save information as to just how much of the instruction had been executed at the time it was interrupted. That would be a lot of extra design work, for rather little performance benefit.

However, in the Intel case, they did make one concession to this idea, in the case of the MOVS instruction family. A MOVS instruction may take so long to execute that it is worth interrupting in the middle, and the Intel engineers did design it this way.

6.4 Glimpse of an ISR

Here is what a simple keyboard ISR might look like, assuming we wish to store the character at a label `keybdbuf` in memory:

```

1      # save interrupted program's EAX and EBX
2      pushl %eax
3      pushl %ebx
4      # get character
5      inb $0x60, %al
6      # copy it to the keyboard buffer
7      movb %al, keybdbuf
8      # acknowledge getting the character
9      inb $0x62, %bl
10     orb $0x20, %bl
11     outb %bl, $0x62
12     andb $0xdf, %bl
13     outb %bl, $0x62
14     # restore interrupted program's EAX, EBX values
15     popl %ebx
16     popl %eax
17     # back to the interrupted program
18     iret

```

The OS/ISR is not running when the interrupt occurs. The key point is that at some times (whenever someone hits a key, in this case) the ISR will need to be run, and from the point of view of the interrupted program, that time is unpredictable. In the example here, each time a character is typed, a different instruction within the program might get interrupted.

6.5 I/O Protection

Linux runs the Intel CPU in **flat protected mode**, which sets 32-bit addressing and enables the hardware to provide various types of security features needed by a modern OS, such as virtual memory. Again for security reasons, we want I/O to be performed only by the OS. The Intel CPU has several modes of operation, which we will simplify here to just User Mode and Kernel (i.e. OS) Mode. The hardware is set up so that I/O instructions such as IN and OUT can be done only in Kernel Mode, and that an interrupt causes entry to Kernel Mode.⁶

6.6 Distinguishing Among Devices

6.6.1 How Does the CPU Know Which I/O Device Requested the Interrupt?

Each I/O device is assigned an ID number, known colloquially as its “IRQ number.” In PCs, typically the 8253 timer has IRQ 0, the keyboard has IRQ 1, the mouse IRQ 12, and so on.

After the I/O device asserts the IRQ line in the bus and the CPU notices, the CPU will then assert the INTA, Interrupt Acknowledge, line in the bus. That assertion is basically a query from the CPU to all the I/O devices, saying, “Whoever requested the interrupt, please identify yourself.” The requesting I/O device then sends its ID number (e.g. 1 in the case of the keyboard) to the CPU along the data lines in the bus, so that the CPU knows that it was device 1 that requested the interrupt.

⁶By setting a special flag in EFLAGS, the OS can allow user programs to execute IN and OUT for specified devices.

6.6.2 How Does the CPU Know Where the ISR Is?

The Intel CPU, like many others, uses **vectored interrupts**. Each I/O device will have its own “vector,” which consists of a pointer to the I/O device’s ISR in memory, plus some additional information.⁷ The pointer and the additional information comprise the “vector” for this I/O device.

All the vectors are stored in an interrupt vector table in memory, a table which the OS fills out upon bootup. Each vector is 8 bytes long, so the vector for I/O device i is located at $c(IDT)+8*i$, where $c()$ means “contents of.” The CPU has an Interrupt Descriptor Table register, IDT. Upon bootup, the OS will point this register to the beginning of the table. So for example, upon bootup, the OS initializes the first 4 bytes at $c(IDT)+8$ to point to the OS’s keyboard device driver (ISR).

Note the hardware/software interaction here: The CPU hardware is what reacts to the interrupt, but the place that the CPU jumps to was specified by the software, i.e. the OS, in the interrupt vector table in memory. The ISR is of course also software.

6.6.3 Revised Interrupt Sequence

We now see that we have to expand the description in Section 6.2 of what the circuitry does when an interrupt occurs:

```

CPU pushes current EFLAGS value on stack
CPU pushes current CS value on stack (irrelevant in 32-bit protected mode)
CPU pushes current PC value on stack
CPU asserts INTA line
CPU reads  $i$  from data bus
CPU computes  $j = c(IDT)+8*i$ 
CPU reads location  $j+4$ , sets interrupt hardware options
CPU reads location  $j$ ; let  $k = c(j)$ 
CPU does  $PC \leftarrow k$ 

```

6.7 How Do PCs Prioritize Interrupts from Different Devices?

Interrupt systems can get quite complex. What happens, for example, when two I/O devices both issue interrupts at about the same time? Which one gets priority? Similarly, what if during execution of the ISR for one device, another device requests an interrupt? Do we suspend the first ISR, i.e. give priority to the second?

To deal with this problem, PCs also includes another piece of Intel hardware, the Intel 8259A interrupt controller chip. The I/O devices are actually connected to the 8259A, which in turn is connected to the IRQ (Interrupt Request) line, instead of the devices being connected directly to the IRQ. The 8259A has many input pins, one for each I/O device.⁸ So, the 8259A acts as an “agent,” sending interrupt requests to the CPU “on behalf of” the I/O devices.

⁷One piece of such “additional information” is the mode in which the ISR is to be run, which typically will be set to Kernel Mode. Later, when the ISR ends by executing an IRET instruction, the mode of the interrupted program will be restored.

⁸These pins are labeled IRQ0, IRQ1, etc., corresponding the IRQ numbers for the devices.

7 DIRECT MEMORY ACCESS (DMA)

If several I/O devices cause interrupts at about the same time, the 8259A can “queue” them, so that all will be processed, and the 8259A can prioritize them. Lots of different priority schemes can be arranged, as the 8259A is a highly configurable, complex chip.

Note that this configuration is done by the OS, at the time of bootup. The 8259A has various ports, e.g. at 0x20 and 0x21, and the OS can set or clear various bits in those ports to command the 8259A to follow a certain priority scheme among the I/O devices.

Today a PC will typically have two 8259A chips, since a single chip can work with only eight devices. The two chips are **cascaded**, meaning that the second will feed into the first. The IRQ output of the second 8259A will feed into one of I/O connection pins of the first 8259A, so that the second one looks like an I/O device from the point of view of the first.

Many other methods of prioritizing I/O devices are possible. For example, we can have the CPU ignore all interrupts until further notice. There is an Interrupt Enable Flag in EFLAGS. The CLI instruction clears it, telling the CPU to ignore interrupts (i.e. skip all Step Ds), while the STI instruction makes the CPU resume responding to interrupts (resume executing a Step D after each Step C).

Designing hardware and software for systems with many I/O devices in which response time is crucial, say for an airplane computer system, is a fine art, and the software aspect is known as **real-time programming**.

7 Direct Memory Access (DMA)

In our keyboard ISR example above, when a user hit a key, the ISR copied the character from the KDP into a register,

```
inb $0x60, %al
```

then copied the register to memory,

```
movb %al, keybdbuf
```

This is wasteful; it would be more efficient if we could arrange for the character to go directly to memory from the KDP. This is called **direct memory access (DMA)**.

DMA is not needed for devices which send or receive characters at a slow rate, such as a keyboard, but it is often needed for something like a disk drive, which sends/receives characters at very high rates. So in many machines a disk drive is connected to a DMA controller, which in turn connects to the system bus.

A DMA controller works very much like a CPU, in the sense that it can work as a **bus master** like a CPU does. Specifically, a DMA controller can read from and write to memory, i.e. assert the MR and MW lines in the bus, etc.

So, the device driver program for something like a disk drive will often simply send a command to the DMA controller, and then go dormant. After the DMA controller finishes its assignment, saying writing one disk sector to memory, it causes an interrupt; the ISR here will be the device driver for the disk, though the ISR won't have much work left to do.

8 Disk Structure

Your disk files consist of sequences of bytes, which in turn consist of bits. Each bit on a disk is stored as a magnetized spot on the disk (a 1 value being represented by one polarity, and a 0 by the opposite polarity). The spots are arranged in concentric rings called **tracks**. Within each track, the spots are grouped into **sectors**, say 512 bytes each. A disk drive will read or write an entire sector at a time; you cannot request it to access just one bit or byte. The disk rotates at a rate of several thousand RPM. A **read/write head** reads/writes the bits which rotate under it.

Each time we do a disk access there are three components of the access time:

- First, the disk's read/write head must be moved to the proper track, an action called the **seek**. This typically takes on the order of tens of milliseconds.
- Then we must wait for the **rotational delay**, i.e. for the desired sector to rotate around to the read/write head.
- Finally, there is the **transfer time**, i.e. the time while the sector is actually being read/written.

Optimization of disk access is a big business. For example, a large banking chain has millions of credit card transactions to deal with every day. If the database program were to not access the disk as quickly as possible, the bank literally might not be able to keep up with the transaction load. Thus careful placement of files on disk is a necessity.

On large systems, several disks (*platters* might be stacked up, one above another. Think for instance of all the Track 3s on the various disks. They will collectively look like a cylinder, and thus even on single-disk systems a synonym for *track* is *cylinder*.

9 USB Devices

The acronym USB stands for **Universal Serial Bus**. The term *bus* here means that many I/O devices can be attached to the same line, i.e. the same USB plug (though on typical home PCs there is no need for this, since there are usually enough USB ports to use separately). The term *serial* refers to the fact that there is only one data line per USB port, as opposed to the 32 or more parallel lines on a system bus.

There are two ways that multiple devices can be attached to the same USB port:

- They can be **daisy-chained**, meaning that, say, device X connects to device Y which in turn is connected to the USB port.
- Several devices can be connected to a **USB hub**. The latter would in turn be connected to a USB port, or into other USB devices or hubs. In this manner, the set of devices connected to a given USB port can have a tree structure.

9 USB DEVICES

At most 127 I/O devices can be connected to a given USB port.

The USB idea really makes things convenient, for several reasons:

- We can connect new devices without having to open up the machine.
- We don't need a lot of expansion slots inside the machine.
- Devices can be **hot-swapped**, i.e. we can connect a new device to a machine which is already running.
- Since we are working with the same general structure, it is easier to write device drivers that work across machines.