Know Your Personal Computer

7. Input-Output Ports

S K Ghoshal

This article briefly describes the input-output port organization of the IBM PC architecture.

Introduction

In IBM PCs and 80X86 architectures, input-output ports are addressed in the same way as memory cells are accessed. An address is produced by the CPU, along with control signals indicating whether the CPU wants to read or write. There is also a line in the control bus (see part 2 of this series) which indicates whether the access is being made to the memory address space or the I/O address space. The on-board decoders (see part 6 of this series) generate the appropriate signals that enable the memory cell or the I/O port as the case may be. The cell/port responds and interchanges data with the CPU over the (same) data bus (that is used for both memory and I/O transactions). See *Figure 1*. Strictly speaking, there is no need for a separate I/O address space for a board-level computer organization to function and interchange data with its own peripheral devices or outside. There are many practical and commercially successful computer organizations in which Input/Output devices are mapped onto the memory address space which is called memory-mapped I/O. However, as the 80X86 microprocessor architecture supports a distinct I/O address space, the IBM PC has a notion of I/O ports. On the motherboard, many chips that comprise the board-level architecture are controlled by the CPU which writes bit-patterns into their appropriate control registers. The CPU can find out the status of any device or of any operation involving that device by reading bit-patterns from appropriate status registers. These control/status registers appear as ports in the I/O address space. Data is interchanged between them and the CPU by a special class of input/output instructions. The I/O

Siddhartha Kumar Ghoshal works with whatever goes on inside parallel computers. That includes hardware, system software, algorithms and applications. From his early childhood he has designed and built electronic gadgets. One of the most recent ones is a sixteen processor parallel computer with IBM PC motherboards.

The previous articles of this series were:

1. Introduction to computers, January 1996.
2. The personal computer hardware, February 1996.
3. The personal computer system software, April 1996.
4. The CPU base architecture, July 1996.
5. The CPU base instruction set and assembly language programming, November 1996.
6. Memory organisation, February 1997.

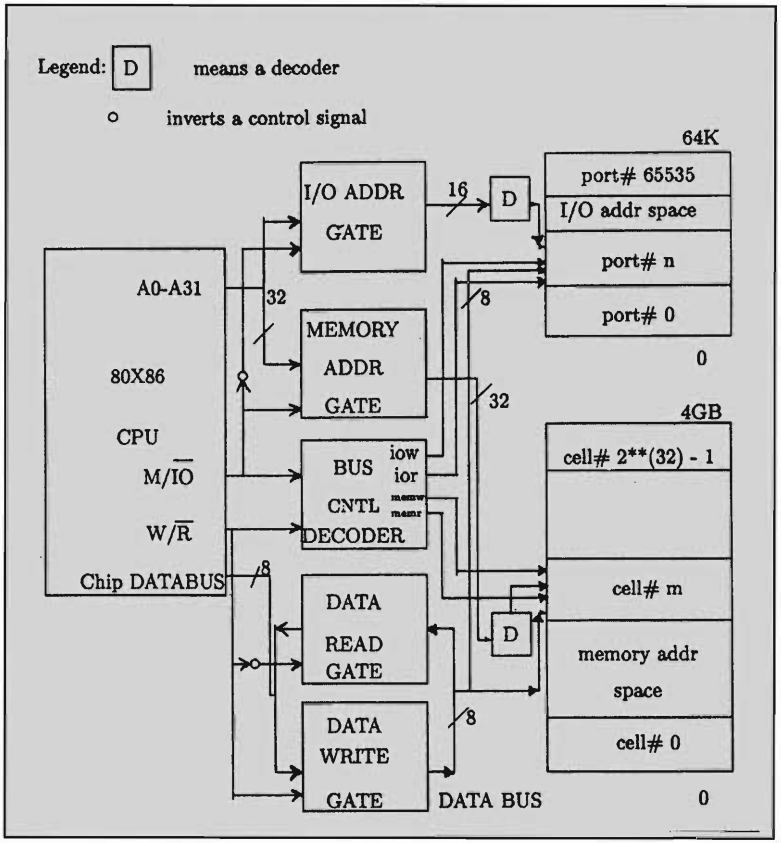

Figure 1 The memory and I/O address spaces are disjoint.

address space is thus disjoint and distinct from the memory address space in both the 80X86 microprocessor architecture and the PC architecture as the latter is based on the former. (See *Figure 1* for the I/O map and the memory map of a typical 80X86 microprocessor architecture.) One can place any hardware register-like object in either the memory address space or the I/O address space by implementing the appropriate decoders. Thus normal memories, memory mapped I/O devices, normal (I/O-mapped) I/O devices and I/O-mapped memory devices can all be supported in the IBM PC motherboard architecture. Each of these interfacing techniques has both standard and innovative uses. In the last article, we had covered normal memory. Before surveying the other interfacing techniques, let us contrast memory and I/O usage in the PC architecture and the programming model used in this architecture.

Contrast Between I/O and Memory

• From one I/O port, a consecutive stream of bytes can be read off and placed in a contiguous span of virtual address space of memory using the DMA controllers (see part 1 of this series). Similarly, the contents of a block of memory can be copied into an I/O port as a stream of bytes. Such transfers *cannot* be done by substituting a memory cell for an I/O port. Thus it is easier to support block devices (I/O devices that transfer data in blocks of bytes are called block devices) using I/O ports. Otherwise each block device controller must have capabilities to do DMA. The design of each such device controller will become very complex and specific to the board-level architecture and the device/ controller pair will be very costly. Thus small-time third party vendors cannot interface their block devices with the board-level architectures if it does not support an I/O address space. Since the IBM PC architecture supports I/O address space, third-party vendors are able to supply peripherals that work as block devices. The driver software of such devices just use the on-board DMA controllers of the PC architecture. Such software is supplied by the third-party vendor or is developed by a fourth party known only to the end-user of the PC. On the other hand, block devices/ controllers/ drivers are proprietary for those architectures that do not support an I/O address space that is distinct from the memory address space of the microprocessor in question.

• Only IN and OUT instructions are allowed on I/O ports. IN moves data from a port to the accumulator. OUT moves data from the accumulator to a port. Both are data transfer instructions of the CPU. No other type of instruction is permitted in 80X86 CPUs when any of its source or destination operand is an I/O port. For example, a data object residing at a port cannot be modified by a CPU instruction. You have to bring it to the accumulator, modify it there and write it back. This is similar to the way RISC CPUs (see part 4 of this series) allow access to the memory. They in fact call the memory as a 'Memory Port'. Thus

it is not surprising that RISC CPUs seldom support a distinct I/O address space. Board-level architectures featuring RISC CPUs manage with memory-mapped I/O.

- Mapping of user-defined objects onto the I/O address space is prohibited. Only objects that are defined by the board-level architecture can be mapped onto I/O ports

- The pattern of usage of the I/O address space and how it gets filled up is dictated by the motherboard architecture and *not* by the user programs. This is where the I/O address space differs markedly from the memory address space.

- To fill up all the I/O address space on any 80X86 CPU, one needs to build an enormously large and complex motherboard architecture featuring that CPU chip with many peripheral devices and controllers. As that has not happened so far, 64KB of I/O address space is enough for all 80X86 microprocessors. The size of the I/O address space does not grow from generation to generation.

Also, I/O is looked upon by both CPU and motherboard designers as a relatively unimportant component of the architecture. They concentrate more on improving the CPU/memory subsystem, while neglecting I/O. (This is like trying to improve the engine of a racing car while neglecting its wheels completely.) This severely restricts the overall capabilities and speed of computers and urgent improvements are needed in I/O subsystems architecture and implementation.[1]

Any modern computer is almost useless if it cannot share and/or interchange information with other computers reliably and at high-speeds. For this one needs good I/O capabilities.

Memory-Mapped I/O

Figure 2 shows a piece of code that makes use of memory-mapped I/O to write 10 lines of the ASCII character 'A' in the video memory of a colour graphics adaptor.[2]

[2] Note that we are using string instructions to write into the memory. This is a typical CISC feature. Refer to part 5 of this series.

Assemble and run this code on your PC. Convince yourself that merely writing data at certain specified locations in the virtual

```
; The following piece of code writes 10 lines
; of 'A' with normal intensity in white
; on a video display using a CGA adaptor
MOV AX, 0B800H      ; Load the segment address of CGA
                      adaptor
MOV ES, AX          ; into the ES register
MOV DI, 0           ; Start writing at offset 0
CLD                 ; Copy string forward
MOV CX, 800         ; 80 chars/line X 10 lines
MOV AX, 0741H       ; Attribute = 07, Character = 'A'
                    ; Repeatedly
REP STOSW           ; Copy AX into memory and increment
                      DI by 2
                    ; Until CX=0
```

Figure 2 Demonstration of memory-mapped I/O.

address space of the 80X86 microprocessor makes the desired display appear on the video terminal. This is typical of memory-mapped I/O. In all CGA adaptors, upon power-up the display is set into the 80×25 mode which is what this program needs in order to work properly. If that is not the case, run the 'setup' utility supplied with your PC. If you do not have a CGA adaptor, either use the appropriate segment address for the type of display adaptor that you actually have on your machine, or make your adaptor emulate a CGA adaptor.

I/O Mapped I/O

When hardware devices are kept in the I/O address space and accessed using I/O instructions, it is called I/O-mapped I/O. The keyboard is interfaced to the IBM PC architecture in this way and the code fragment listed in *Figure 3* uses it to return the scan code and keyboard status to its C caller. How the call-return and parameter passing mechanisms work between this code fragment and its C caller has been explained in part 5 of this series. Run this code after writing an appropriate C caller (which keeps calling this routine and printing its return value in an infinite loop), compiling it and linking the object code with

Figure 3 A C-callable assembly language program to read the keyboard data and status ports.

```
.model small ; the virtual address space is 64KB wide
.code ; executable code follows in this module
public _GETKBD ; that can be called from any C program
_GETKBD PROC NEAR ; Remember to use a small memory model.
See part 5.
IN AL, 60H ; Read in the Keyboard data.
XCHG AH, AL ; Put it in MSB of AX.
IN AL, 61H ; Read in the Keyboard status in LSB of AX.
RET ; Go back to C caller with return value in AX.
_GETKBD ENDP ; The assembly language function ends here.
end ; the assembly module ends here
```

what you get after assembling the code fragment in *Figure 3*. Keep pressing various keys and see how the status and the scan code keep changing. Note that we are bypassing the low-level system software here and accessing the I/O mapped keyboard data port and control port directly.

I/O Mapped Memory

This is used for one or more of the following reasons:

- Some memories are inside chips that constitute the board-level architecture of the IBM PC. Thus they are better suited to be mapped onto the I/O address space.

- There is always room in the I/O address space of the CPU, even when it is a large and complicated on-board architecture, for making it more powerful by adding more hardware devices.

- The high-level operating systems and the user-written application programs cannot and do not reserve blocks of storage within the I/O address space. Therefore, if a device is mapped onto the I/O address space, the operating system and the application will run on that architecture, without any I/O address clashing.

What is a Scan Code

There is an Intel 8048 micro-controller below every IBM PC keyboard. It is actually an embedded controller (see part 1 of this series) which keeps on scanning the keyboard for any key pressed . When it finds one, it determines its geometric location on the keyboard, encodes it as a 'scan code', writes that code into the port 60H and interrupts (see part 3 of this series) the CPU. The CPU then converts the scan code into the ASCII code of the character, using a lookup table that can be easily changed by a product developer as it is in firmware (see part 2 of this series). Thus one can easily change the location of the keys in the IBM PC keyboard, just by putting new stickers on the key-tops and adjusting the lookup table. Different types of keyboards (the ones for Japanese, Chinese, European Languages etc. and special application-specific keyboards in which the key-tops are painted with symbols, rather than alphabets) are supported this way without having to change the keyboard electronics. For example, one can designate (equivalent of ASCII) codes for Dosa, Idli, Tea, Coffee etc for the customer-interface of a restaurant which one is automating using an IBM PC. Just paint the key-tops with the pictures of these items and you have the keyboard ready to accept customer orders. The keyboards we normally use are American English keyboards. The code in *Figure 3* allows one to see what is going on between it and the IBM PC, without interfering with the working of either. As one can infer, the indirect way of generating the ASCII codes for US English from scan codes makes it slightly slower than generating ASCII codes by the 8048 itself. However, one gains a lot of flexibility. And in any case for keyboards, the speed is limited by the maximum rate at which one can type in information, which is very very slow compared to that of the 8048 or the 80X86 CPU. Thus the loss of speed cannot be perceived whereas the flexibility is appreciated.

- By using one port to index into the device memory and another one to serve as the data port to that entire device memory, one can bring in asynchronism between the 80X86 CPU and the (possibly much slower) device memory. All devices cannot keep pace with the improvements in the 80X86 CPU. However board-level products must be made to work reliably as soon as any new 80X86 CPU is announced. Thus this feature is crucial to the advent of the IBM PC.

A large number of peripherals are interfaced in the asynchronous mode with the CPU using the I/O address space. This is because they are slow, geographically remote, must consume less power and thus must not be accessed at the speed with which the CPU accesses memories. A typical example is the battery-powered real-time clock and configuration RAM (RTC) chip whose device memory is accessed via the I/O address space using

Figure 4 A function to read the RTC.

```
.model small  ; the virtual address space is 64KB wide
.code ; executable code follows in this assembly language module
public _GDATE; public near procedure  callable by C modules
_GDATE PROC NEAR ; thread of control enters at this point
PUSH BP; Save caller's context
MOV  BP, SP; Fix the context of this instance
MOV  AL, 07 ; index value for date of the month
OUT  70H, AL ; Write that into the index register
IN   AL, 71H ; Read the date from the RTC data register
MOV  AH, 0  ; Zero out MSB of accumulator
POP  BP; Restore caller's context
RET;  Return thread of control to caller
```

an index(address 70H) and a data register (address 71H), both I/O mapped. The battery keeps the real-time clock ticking and the information in the configuration RAM intact for many years. If the RTC chip had to work at the CPU speed in a modern 80X86 architecture, the battery would have drained out in a few hours. *Figure 4* shows how to access the RTC data. A C-callable assembly language function (see part 5 of this series) is written to return the date of a month. Assemble, link and run it as was done for the program in part 5. Note that the address space where the code of *Figure 4* is going to load and run from is the virtual address space of the 80X86 CPU. That space is populated by regular (memory-mapped) memory.

In subsequent articles we shall deal with peripheral devices that are accessed in the IBM PC architecture via the I/O address space and how the low-level operating system of the IBM PC (called BIOS- Basic Input Output System) initializes and drives them on behalf of the high-level operating systems (see part 3 of this series). Comments about this article are welcome.

Address for Correspondence
S K Ghoshal
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560 012, India
email:
ghoshal@serc.iisc.ernet.in
Fax: (080) 334 1683

Suggested Reading

- ***The IBM PC/XT Technical Reference Manual.*** **IBM Corporation, 1984.**
- **V C Hamacher and others. *Computer Organization.* Third Edition. McGraw Hill International, 1990.**