Input-Output (I/O) Organizations

Overview:

What constitutes input/output in a computer system? It is clear that a keyboard is an input device and a printer is an output device, but what about an internal disk drive? The answer is that a disk is an I/O device not because information comes from or goes to the world outside the computer, but because it moves through an I/O subsystem that connects the processor and memory to devices having very different physical characteristics, especially when it comes to speed and timing.

The *I/O subsystem* provides the mechanism for communications between the CPU and the outside world. It must manage communications that are not only asynchronous with respect to the CPU, but also have widely ranging data rates.

Processor and memory live in an artificial world regulated entirely by a master clock. Even things like waits for dynamic RAM refresh are limited to a few whole clock cycles. On the other hand, print timing may be governed by the mechanical movement of print hammers; keyboard input depends on human muscles and thought, and internal disk timing is bound to the real world through both read/write head movement and rotational delay. Data arriving from a network connection will likewise arrive unexpectedly. Furthermore, in this latter case, the data must be captured by the computer as it arrives.

It provides a systematic means of controlling interaction with the outside world and provides the operating system with the information it needs to manage I/O activity effectively.

Three factors must be considered in designing an I/O subsystem:

1. Data location: device selection, address of data within device
2. Data transfer: amount, rate, to or from device
3. Synchronization: output only when device ready; input only when data available

All of the three things needed to carry out an I/O data transfer are dependent on the type of I/O device involved, and it is important to move toward some standardization-a *contract* between CPU and I/O device defining the data structure(s) at the device interface. An easy step is for the device interface to handle the (messy) details such as assembling bits into words, doing low-level error detection and correction, and accepting or providing words in word-sized registers located at defined addresses within the I/O space. This places the responsibility for low-level data formatting on the I/O device, thus presenting a uniform interface to the CPU. Thus to the CPU and the I/O device, the device interface consists of a set of data and control registers within the I/O address space. The contract between CPU and I/O device is specific to the device, and is generally spelled out in a written document. In a given I/O device specification, some registers, such as those holding data provided by the device, will be read-only. Others, such as those that are designed to receive data from the CPU, will be write-only. Some, such as device status registers, will be read/write. Even synchronization can sometimes be cast in the form of data transmitted and received. A bit in an output register may tell a device to start a specific operation, and a bit in an input register may signal completion of the operation to the processor. This synchronization method is known as ***programmed I/O.*** If the speed of an I/O device is in the right range, neither too fast for the processor to read and write the signaling bits nor too slow for the processor to wait for its activity, this form of signaling may be sufficient. Different forms of synchronization are appropriate for different device speed ranges. Long and variable latencies, or response times, bring a need for **I/O *interrupts*** to support synchronization, and high-bandwidth bursts of data transmission are synchronized using D***irect Memory Access***, or DMA.

I/0 Bus Structures:

Reducing all three requirements of location, data transfer, and synchronization to moving control or data information to and from I/O registers puts the focus on the data transmission path, or bus, between processor and I/O device

registers. Only two I/O operations are needed, input from and output to a specified I/O register. One parameter of these operations is an I/O register address, and a second might identify a processor register as source or destination for the information. Input looks very much like a load instruction and output like a store. An I/O bus thus behaves much like a memory bus. It could differ in timing and synchronization, or it could leave those differences to the individual I/O device interfaces making up the I/O subsystem. Different computer systems use different degrees of separation between I/O data transmission and memory data transmission, as shown in Figure.

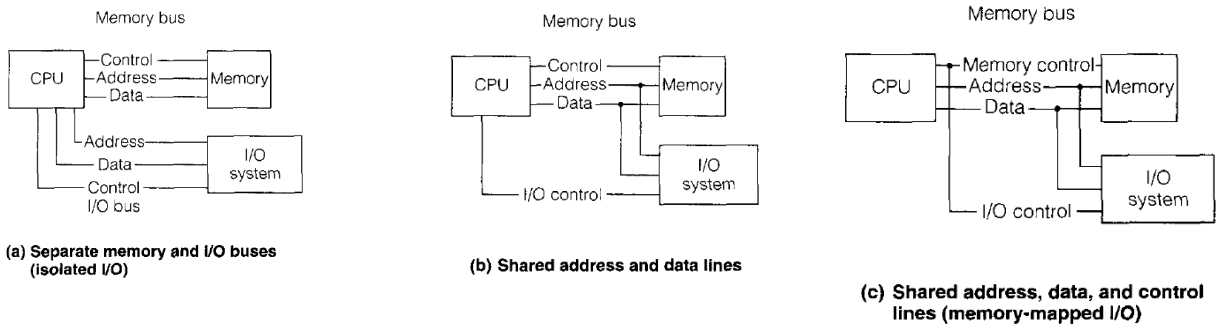


Fig1.: Independent and Shared Memory **and I/O Buses**

The system of Figure 1a), sometimes referred to as *isolated I/O,* has one set of address, data, and control wires for an I/O bus and another set for a memory bus. The system of Figure 1 b) shares address and data wires between I/O and memory buses but has different control signals for read, write, input, and output. This is sometimes called *shared I/0.* Since there are usually many fewer device I/O registers in a system than memory addresses, not all address signals may be used in connection with input and output operations.

Finally, in *memory-mapped I/U,* shown in Figure 1c), a single bus is used for both 1/0 and memory. The diagram is obtained by simply combining I/O and memory control connections. A range of memory addresses is reserved for I/O registers, and these registers are read and written using standard load and store instructions. Memory-mapped I/O is common in modern processors. It has two primary motivations: data transfer to and from the processor is standardized, and the number of connections to the processor chip or board is reduced.

Three principal I/O techniques:

1. **Programmed I/O**: in which I/O occurs under the direct and continuous control of the program requesting the I/O operation;
2. **Interrupt-driven I/O**: in which a program issues an I/O command and then continues to execute, until it is interrupted by the I/O hardware to signal the end of the I/O operation; and
3. **Direct Memory Access (DMA): in** which a specialized I/O processor takes over control of an I/O operation to move a large block of data.

Programmed I/O: *(reference: book-COMPUTER ORGANIZATION AND ARCHITECTURE DESIGNING FOR PERFORMANCE, EIGHTH EDITION, by William Stallings)*

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register (Figure 7.3).The I/O module takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, it is the responsibility of the processor periodically to check the status of the I/O module until it finds that the operation is complete.

To explain the programmed I/O technique, we view it first from the point of view of the I/O commands issued by the processor to the I/O module, and then from the point of view of the I/O instructions executed by the processor.

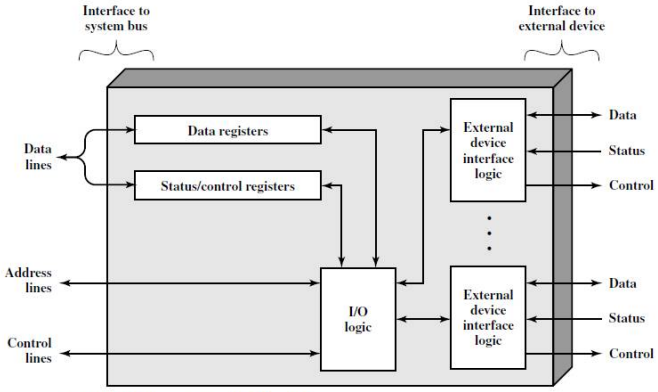


Figure 7.3 Block Diagram of an I/O Module

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:

• Control: Used to activate a peripheral and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record. These commands are tailored to the particular type of peripheral device.

• Test: Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know that the peripheral of interest is powered on and available for use. It will also want to know if the most recent I/O operation is completed and if any errors occurred.

• Read: Causes the I/O module to obtain an item of data from the peripheral and place it in an internal buffer (depicted as a data register in Figure 7.3).The processor can then obtain the data item by requesting that the I/O module place it on the data bus.

• Write: Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit that data item to the peripheral.

Figure 7.4a gives an example of the use of programmed I/O to read in a block of data from a peripheral device (e.g., a record from tape) into memory. Data are read in one word (e.g., 16 bits) at a time. For each word that is read in, the processor must remain in a status-checking cycle until it determines that the word is available in the I/O module's data register.

This flowchart highlights the main disadvantage of this technique:
it is a time-consuming process that keeps the processor busy needlessly.

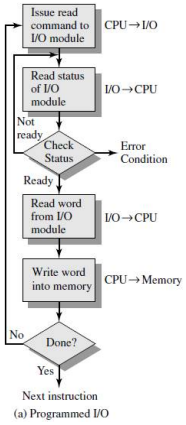


Figure 7.4 a)of programmed I/O

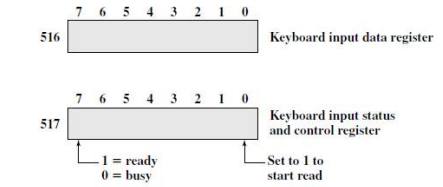
I/O Instructions:

With programmed I/O, there is a close correspondence between the I/O-related instructions that the processor fetches from memory and the I/O commands that the processor issues to an I/O module to execute the instructions. That is, the instructions are easily mapped into I/O commands, and there is often a simple one-to-one relationship. The form of the instruction depends on the way in which external devices are addressed.

Typically, there will be many I/O devices connected through I/O modules to the system. Each device is given a unique identifier or address. When the processor issues an I/O command, the command contains the address of the desired device. Thus, each I/O module must interpret the address lines to determine if the command is for itself.

When the processor, main memory, and I/O share a common bus, two modes of addressing are possible: memory mapped and isolated. With memory-mapped I/O, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both memory and I/O devices. So, for example, with 10 address lines, a combined total of 210 1024 memory locations and I/O addresses can be supported, in any combination.

With memory-mapped I/O, a single read line and a single write line are needed on the bus. Alternatively, the bus may be equipped with memory read and write plus input and output command lines. Now, the command line specifies whether the address refers to a memory location or an I/O device.The full range of addresses may
be available for both.Again, with 10 address lines, the system may now support both 1024 memory locations and 1024 I/O addresses. Because the address space for I/O is isolated from that for memory, this is referred to as isolated I/O.



ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

(a) Memory-mapped I/O

ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load I/O	5	Initiate keyboard read
201	Test I/O	5	Check for completion
	Branch Not Ready	201	Loop until complete
	In	5	Load data byte

(b) Isolated I/O

Figure 7.5 Memory-Mapped and Isolated I/O

Figure 7.5 contrasts these two programmed I/O techniques. Figure 7.5a shows how the interface for a simple input device such as a terminal keyboard might appear to a programmer using memory-mapped I/O. Assume a 10-bit address, with a 512-bit memory (locations 0–511) and up to 512 I/O addresses (locations 512–1023). Two addresses are dedicated to keyboard input from a particular terminal. Address 516 refers to the data register and address 517 refers to the status register, which also functions as a control register for receiving processor commands. The program shown will read 1 byte of data from the keyboard into an accumulator register in the processor. Note that the processor loops until the data byte is available.

With isolated I/O (Figure 7.5b), the I/O ports are accessible only by special I/O commands, which activate the I/O command lines on the bus. For most types of processors, there is a relatively large set of different instructions
for referencing memory. If isolated I/O is used, there are only a few I/O instructions. Thus, an advantage of memory-mapped I/O is that this large repertoire of instructions can be used, allowing more efficient programming. A disadvantage is that valuable memory address space is used up. Both memory-mapped and isolated I/O are in common use.

INTERRUPT-DRIVEN I/O:

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line.The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts. When the interrupt from the I/O module occurs, the processor saves the context (e.g., program counter and processor registers) of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program it was working on (or some other program) and resumes execution.

Figure 7.4b shows the use of interrupt I/O for reading in a block of data. Compare this with Figure 7.4a. Interrupt I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

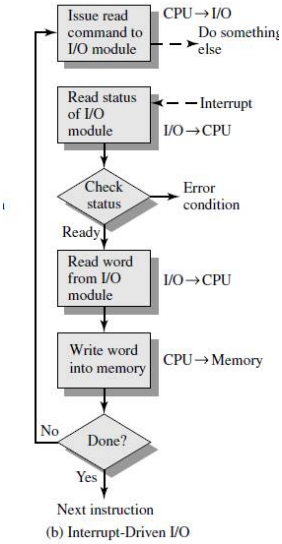


Fig. 7.4 b)

Interrupt Processing:

Let us consider the role of the processor in interrupt-driven I/O in more detail. The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software. Figure 7.6 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for an interrupt, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is (a) the status of the processor, which is contained in a register called the program status word (PSW), and (b) the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto the system control stack.2
5. The processor now loads the program counter with the entry location of the interrupt-handling program that will respond to this interrupt. Depending on the computer architecture and operating system design, there may be a single

program; one program for each type of interrupt; or one program for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information

may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information. Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, the result is that control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the "state" of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So, all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack.

Figure 7.7a shows a simple example. In this case, a user program is interrupted after the at location N. The contents of all of the registers plus the address of the next instruction (N +1) are pushed onto the stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.

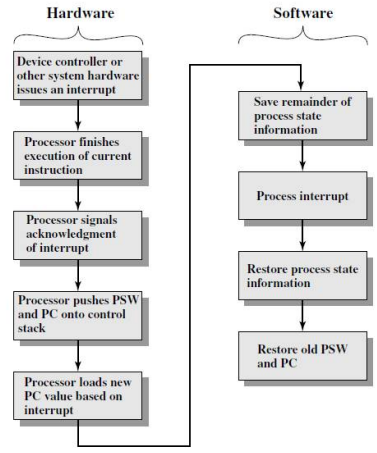


Figure 7.6 Simple Interrupt Processing

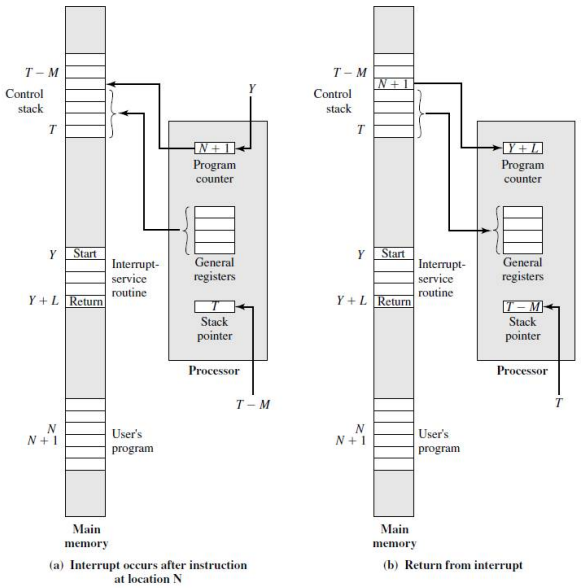


Figure 7.7 Changes in Memory and Registers for an Interrupt

7. The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.

8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 7.7b).

9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

Design Issues:

Two design issues arise in implementing interrupt I/O. First, because there will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt? And second, if multiple interrupts have occurred, how does the processor decide which one to process?

Let us consider device identification first. Four general categories of techniques are in common use:

- Multiple interrupts lines
- Software poll
- Daisy chain (hardware poll, vectored)
- Bus arbitration (vectored)

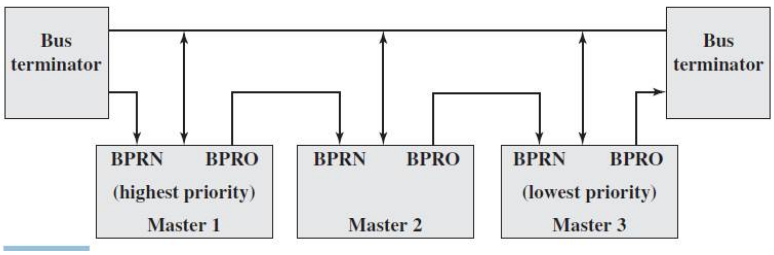
Multiple interrupt lines:
The most straightforward approach to the problem is to provide **multiple interrupt** lines between the processor and the I/O modules. However, it is impractical to dedicate more than a few bus lines or processor pins to interrupt lines. Consequently, even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus, one of the other three techniques must be used on each line.

Software poll:

One alternative is the software poll. When the processor detects an interrupt, it branches to an interrupt-service routine whose job it is to poll each I/O module to determine which module caused the interrupt. The poll could be in the form of a separate command line (e.g., TESTI/O). In this case, the processor raises TEST I/O and places the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt. Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module. Once the correct module is identified, the processor branches to a device-service routine specific to that device.

Daisy chain (hardware poll, vectored):
The disadvantage of the software poll is that it is time consuming. A more efficient technique is to use a daisy chain, which provides, in effect, a hardware poll. For interrupts, all I/O modules share a common interrupt request line. The interrupt acknowledge line is daisy chained through the modules. When the processor senses an interrupt, it sends out an interrupt acknowledge. This signal propagates through a series of I/O modules until it gets to a requesting module. The requesting module typically responds by placing a word on the data lines. This word is referred to as a vector and is either the address of the I/O module or some other unique identifier. In either case, the processor uses the vector as a pointer to the appropriate device-service routine. This avoids the need to execute a general interrupt-service routine first. This technique is called a vectored interrupt.

An example of a daisy-chain configuration is shown in Figure .



//Figure indicates a distributed arbitration scheme that can be used with an obsolete bus scheme known as Multibus I. Agents are daisy-chained physically in priority order. The left-most agent in the diagram receives a constant bus priority in (BPRN) signal indicating that no higher-priority agent desires the bus. If the agent does not require the bus, it asserts its bus priority out (BPRO) line. At the beginning of a clock cycle, any agent can request control of the bus by lowering its BPRO line. This lowers the BPRN line of the next agent in the chain, which is in turn required to lower its BPRO line. Thus, the signal is propagated the length of the chain. At the end of this chain reaction, there should be only one agent whose BPRN is asserted and whose BPRO is not. This agent has priority. If, at the beginning of a bus cycle, the bus is not busy (BUSY inactive), the agent that has priority may seize control of the bus by asserting
the BUSY line.

Bus arbitration (vectored):

There is another technique that makes use of vectored interrupts, and that is bus arbitration. With bus arbitration, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places its vector on the data lines. The aforementioned techniques serve to identify the requesting I/O module.

They also provide a way of assigning priorities when more than one device is requesting interrupt service. With multiple lines, the processor just picks the interrupt line with the highest priority. With software polling, the order in which modules are polled determines their priority. Similarly, the order of modules on a daisy chain determines their priority.