Memory, input, output and interrupts.

In single-bus machines, the same bus serves both as a *memory bus* and as an *I/O bus*, though this isn't the only way to arrange things. However, if this is the case, I/O devices are treated as if they are simple memory locations. For example, in a 32-bit Windows machine, the virtual memory space is 4GBytes, and the first 3 GBytes are assigned to programs and their data. The last GByte of addresses are assigned to be associated with peripherals. Locations in this last 1GByte space are intercepted, and information and device status reports are communicated to the CPU (over the memory bus) as if the CPU has just accessed normal memory. *This removes much of the need for special I/O instruction to be present in the instruction set.*

I/O transfers may be distinguished from memory read and write operations by including a special I/O control line on the bus, **IOC**, joining such signals as R/W', WMFC and MFC. This new control line (**IOC**) is activated by the CPU during a read or a write operation that is intended for an I/O device. The memory unit responds to commands on the bus when the I/O line is inactive, and I/O devices respond when it is active.

Memory Mapping

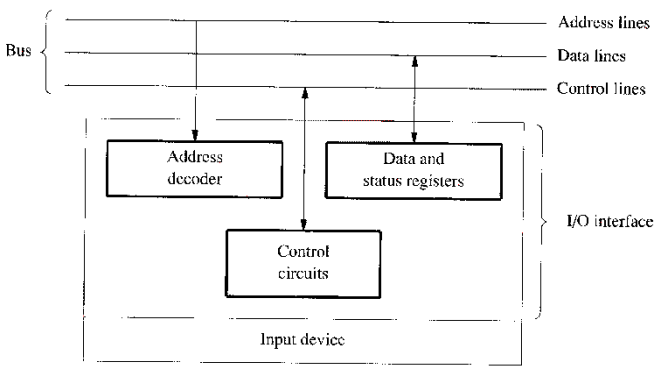
In summary, this arrangement for single-bus machines (and others if appropriate) serves to identify and communicate with I/O devices by assigning them unique addresses within the **memory-mapped address** space of the computer. Thus, it becomes possible to access I/O devices in the same way as *any other memory location.* This arrangement is known as memory-mapped I/O. Neither special "IN" and "OUT" instructions nor a set of special I/O control lines on the bus are needed. Instead, any instruction that moves data to or from a memory location can be used to transfer data to or from an I/O device. For example, if the variable "INBUF" contains the address of the input buffer associated with the keyboard, then the 68000 instruction:

MOVE.B INBUF, MEM

reads one byte (that's the ".B" part) from INBUF and deposits it into memory location named MEM.

The use of memory-mapped I/O offers considerable flexibility in handling I/O operations, because any machine instruction and *addressing mode* that can be used to deal with conventional memory operands can also refer to an I/O device. It is customary, though not necessary, to assign a contiguous portion of the address space of the machine to I/O devices. (See the Windows 32-bit machine details above).

The 68000 uses memory-mapped I/O. Early Intel microprocessors had special I/O instructions and a separate 16-bit address space for I/O devices, so as I said, this isn't the only way to arrange things.



I/O interface for an input device.

The figure illustrates the arrangement required to connect an I/O device to the bus of a computer. The address decoder allows the device to recognize its address when this address is placed on the address bus by the CPU. The data register is used to hold data to be transferred to the CPU from an input device or to receive data from the CPU for transfer to an output device.

I/O devices often have one or more status registers that contain information relevant to their operation. Such registers must also be connected to the data bus and assigned

unique addresses. The address decoder, the data and status registers, and the control hardware required to coordinate I/O transfers constitute the device interface circuit.

Registers for data may be insufficient. For devices like a hard drive, a copious amount of RAM memory is provided to buffer data at electronic speeds, and in sufficient quantity to cope with "block" transfers. Same for printers.

Direct Memory Access

In the discussion above, we assumed that machine instructions such as "IN", "OUT", or "MOVE" could be used to transfer data to or from an I/O device. If the CPU executes such instructions, then it is performing *"program-controlled I/O"* which works, but subtracts from the CPU's availability for more productive work as this I/O effort can be largely delegated to other circuits.

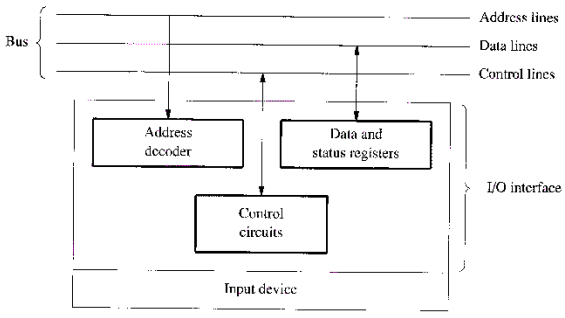
When large blocks of data need to be transferred at high speed, an alternative approach may be used. A special control unit may be provided to enable transfer of a block of data directly between an external device and the main memory, *without continuous intervention by the CPU*. This approach is called ***direct memory access***, or DMA. The DMA has considerable buffer memory.

Program-controlled I/O is unsuitable for high-speed data transfer for two reasons:

1. In program-controlled I/O considerable overhead is incurred, because several program instructions have to be executed for each data word transferred between the external device and the main memory.

2. Many high-speed peripheral devices have a synchronous mode of operation. That is, data transfers are controlled by a clock of fixed frequency, independent of the CPU. Reading single bytes is out of the question.

The problems encountered in operating high-speed devices can be overcome by incorporating all functions performed by I/O routines in a **hardwired controller**. The purpose of this controller is to enable direct data transfer between the device and the main memory *without involvement of the CPU*.



I/O interface for an input device.

Inspection of the figure suggests that such a DMA controller requires two counter registers, one for generating the memory address and the other for keeping track of the word count. A third register is needed to store a command specifying the function to be performed. (R/W', for example). For more complex devices, such as disk drives, other registers may also be required to control their operation. Since controller registers must be accessible for initialization by the CPU under program control, they should be connected to the bus and assigned unique addresses, as in the case of any other I/O device interface.

A DMA controller can be used in conjunction with two or more I/O devices, say a disk drive and a high-speed printer. In this case, the controller is said to provide two DMA *channels*. The registers needed to store the memory address, the word count, and so on are duplicated, so that one set can be used with each device. A connection is also provided for each channel between the DMA controller and one of the I/O devices.

To start a DMA transfer of data between the disk and the main memory, a program writes the following information into the registers of the DMA channel assigned to the disk:

- Main Memory address – where to begin the MM read or write
- Word/Byte count -- the number of words or bytes that should be transferred
- Address of the *start* of data on the disk – you write starting here, or you read starting here
- Function to be performed (Read or Write)

The DMA controller then proceeds *independently* to implement the function specified. It uses its connection to the disk unit to synchronize its operation with that of the disk. When the DMA transfer is completed, this fact is recorded in the status register of the DMA controller. The status register will also contain information indicating whether the transfer took place correctly or if some errors were encountered.

For a *write* to the disk, the DMA will fill its buffer *at high speed with data from the main memory* and then drop the data down to the hard drive at a speed compatible with that disk. Thus we bridge a speed "gap".

While the DMA transfer is taking place, the program that requested the transfer cannot continue, because it must wait for the results of the transfer. However, the CPU *can* be used to execute another program. This is efficient, and the operating system takes care of this.

After the DMA transfer is completed the CPU may switch back to the program that requested the transfer. *It is the responsibility of the operating system to suspend the execution of one program and to start another.* It is also the operating system that initiates the DMA operation when requested to do so by a program. When the transfer is completed, the DMA controller informs the CPU by means of a control signal on the bus. (Add **INTR** and **INTRA** to the set of control signals in the memory bus.)

The controller activates this signal at the same time it sets the Ready bit in its status register. The interrupt mechanism and the way it is used to coordinate DMA and other I/O operations will be discussed later.

Note that a conflict situation may arise if both the CPU and a DMA controller try to access the main memory at the same time. To resolve this conflict, a special circuit called the *bus arbiter* (don't worry about it – it's just a circuit) is provided to coordinate the activities of all devices requesting memory transfers. The arbiter implements a priority system, as will be described later.

Memory accesses by the CPU and DMA controllers are interwoven, with top priority given to transfers involving synchronous, high-speed peripherals such as disk and fast printers or graphics cards. Considering that in most cases the CPU originates the majority of memory access cycles, the DMA controller can be regarded as "stealing" memory cycles from the CPU. Hence, this interweaving technique is usually referred to as ***cycle stealing***.

Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as *burst mode*.

Interrupts

A computer must have some means of coordinating its activities with those of the external devices connected to it. For example, when accepting characters from a keyboard, the computer needs to know when a new character has been typed. Similarly, during an output operation, it should not send a character to a printer until the printer is online and ready to accept it.

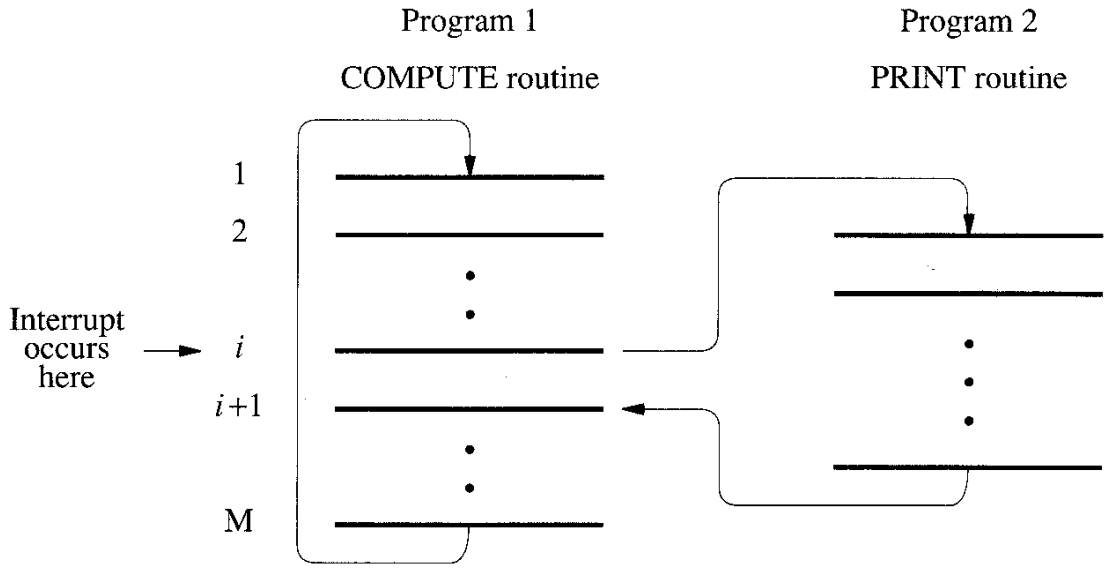
The current status of each input or output device connected to a computer is indicated by one or more bits of information. The most important of these bits, usually referred to as the Ready bit, is set whenever the device is ready to participate in a new transfer operation. A program should survey the device by testing its status bits before attempting an I/O operation.

A printer, as an example, will have many status bits in use: some are: “online”, “ready”, “paper out”, “toner/ink low”, “paper jam”, “print buffer full”, “paper output tray full” and so on.

Either the CPU regularly “polls” its devices to see if they need attention, or the CPU instead agrees to be ***interrupted*** when an I/O device needs to participate in an I/O operation. Clearly, during a period of “polling”, the CPU is not performing any useful computation.

There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the CPU when it becomes ready. It can do so by sending a hardware signal called an ***“interrupt”*** to the CPU. At least one of the bus control lines, called an interrupt-request line, is usually dedicated for this purpose. (Reminder: add this to the control lines in the memory bus too!)

The CPU can instruct an I/O device to activate this line at the same time that it sets the Ready bit in its status register. Since the CPU is no longer required to continuously check the status of external devices, the polling period can be utilized to perform other useful functions. Indeed, by using interrupts, such polling periods can ideally be eliminated.



Transfer of control through the use of interrupts.

Consider a task that requires some computations to be performed and the results to be printed on a line printer. This is followed by more computations and output, and so on. Let the program consist of two routines, COMPUTE and PRINT. Assume that COMPUTE produces ***n*** lines of output, to be printed by the PRINT routine.

The required task may be performed by repeatedly executing first the COMPUTE routine and then the PRINT routine. Assuming that the printer accepts only one line of text at a time, then the PRINT routine must send one line of text, wait for it to be printed, then send the next line, until all the results have been printed. The disadvantage of this simple approach is that the CPU spends a considerable amount of time waiting for the printer to become ready. It’s just too inefficient.

If it is possible to overlap printing and computation, that is, to execute the COMPUTE routine while printing is in progress, a faster overall speed of execution will result. This may be achieved as follows. First, the

COMPUTE routine is executed to produce the first ***n*** lines of output. Then, the PRINT routine is executed to send the first line of text to the printer. At this point, instead of waiting for the line to be printed, the printer may be ignored and execution of the COMPUTE routine continued. Whenever the printer becomes ready, it alerts the CPU by sending an interrupt-request signal (**INTR** or **IRQ**). In response, the CPU interrupts execution of the COMPUTE routine and transfers control to the PRINT routine. The PRINT routine sends the second line to the printer, which is again ignored. Then the interrupted COMPUTE routine resumes execution at the point of interruption. This process continues until all ***n*** lines have been printed. This is a simple minded situation, but it illustrates the point that we need to keep the CPU busy doing useful work, and not just inconveniently waiting for a very slow device.

So, the PRINT routine will be restarted whenever the next set of ***n*** lines is available for printing. If COMPUTE takes longer to generate ***n*** lines than the time required to print them, the CPU will be performing useful computations all the time. Performance is enhanced.

The example above is intended to introduce just the concept of interrupts. The routine executed in response to an interrupt request is called the interrupt-service routine **(ISR)**. This is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine or function calls.

Assume that an interrupt request arrives during execution of instruction **i** in the figure. What happens? What is the sequence of events?

1. The CPU first completes execution of instruction **i**. FINISH THE CURRENT INSTRUCTION!
2. Then, it loads the program counter with the address of the first instruction of the interrupt service routine **(ISR)**. This it typically gets from a *vectored interrupt table* which, for each device, holds the address of the correct **ISR.** Devices are numbered, so this is an easy table to construct, having two fields: device number + staring address of its **ISR.**
3. After execution of the interrupt-service routine, the CPU has to come back to instruction **i + 1**. Therefore, when an interrupt occurs, the current contents of the PC, which point at instruction **i + 1**, have to be put in temporary storage. A Return-from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction **i + 1**. In most computers, the return address is saved on the processor stack.

Storing just the return address **i + 1** is wholly inadequate, and would not allow for the resumption of the running program. This is because the **ISR** runs on the same CPU as the main program *and will use the general and special purpose registers, over-writing partial results and any data which the main program had generated*. Therefore, it is necessary to stack nor only the PC, but also:

- The general registers
- Many of the special purpose registers
- The flags, NVCZ included

It is left for the reader to decide what needs to be stacked in order to successfully resume an interrupted program. For each register in the table below, put yourself into the position of a running program and decide whether you will need to store that register in order to resume running. Store/stack or not? (Assume the one-bus architecture in answering). Saying "Yes" to all is *not* the answer.

Register	Stack, Y or N
General Registers (all)	
MAR	
MDR	
IR	

PC	
Flags	
Buffer Y	
Buffer Z	
Function select lines	
Source (SRC)	
Dest (DEST)	
Temp (TMP)	

We should note that as part of handling interrupts, the CPU must inform the device that its request has been recognized, so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus. An interrupt-acknowledge signal, **INTRA,** used in some – most, but not all -- of the interrupt schemes to be discussed later, serves this function. A common alternative is to have the transfer of data between the CPU and the I/O device interface accomplish the same purpose. In this case, the execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface *implicitly informs the device that its interrupt request has been recognized.* (There's usually more than one way to accomplish something).

So far, treatment of an interrupt-service routine is very similar to that of a subroutine. An important departure from this similarity should be noted. A subroutine performs a function *required by the program from which it is called*. However, the interrupt-service routine (**ISR)** may not have anything in common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users. Therefore, before starting execution of the interrupt-service-routine, the CPU should save, along with the contents of the PC, any information that may affect execution after return to the interrupted program. In particular, the CPU should save the register that includes the condition codes and any other status indicators at the time of interruption. Upon return from the interrupt-service routine, the CPU reloads the condition codes from their temporary storage location. This enables the original program to resume execution without being affected in any way by the occurrence of the interrupt, except, of course, for the time delay.

The contents of CPU registers other than the program counter and the processor status register may or may not be saved automatically by the interrupt-handling mechanism. If the CPU does save register contents before entering the interrupt-service routine, it must also restore them before returning from the **ISR** back to the interrupted program. The process of saving and restoring registers involves a number of memory transfers that create a time overhead for every interrupt accepted by the CPU.

In a computer that does not automatically save all necessary register contents following an interrupt, the interrupt-service routine should save the contents of any CPU register that it needs to use. The saved data should be restored to their respective registers before returning to the interrupted program.

In order to minimize the interrupt overhead, some computers provide two types of interrupts. One saves all relevant register contents in the stack, and the other does not. A particular I/O device may use either type, depending upon its response time requirements. An example of this approach is found in Motorola's 6809 microprocessor. Another interesting approach is to provide duplicate sets of CPU registers, so that a different set of registers can be used when servicing interrupt requests – and **ISR** would switch to the second set of registers.

"**Interrupt Latency**" is a term applied to the time taken to finish the instruction which was running when the interrupt occurred. The peripheral set must be able to tolerate this slight delay before receiving attention from the **ISR**. This is the minimal delay figure, because other restrictions, such as the interrupt being "disabled", can add to it.

An interrupt is more than a simple mechanism for coordinating I/O transfers. In a general sense, interrupts enable transfer of control from one program to another to be initiated by an event external to the computer.

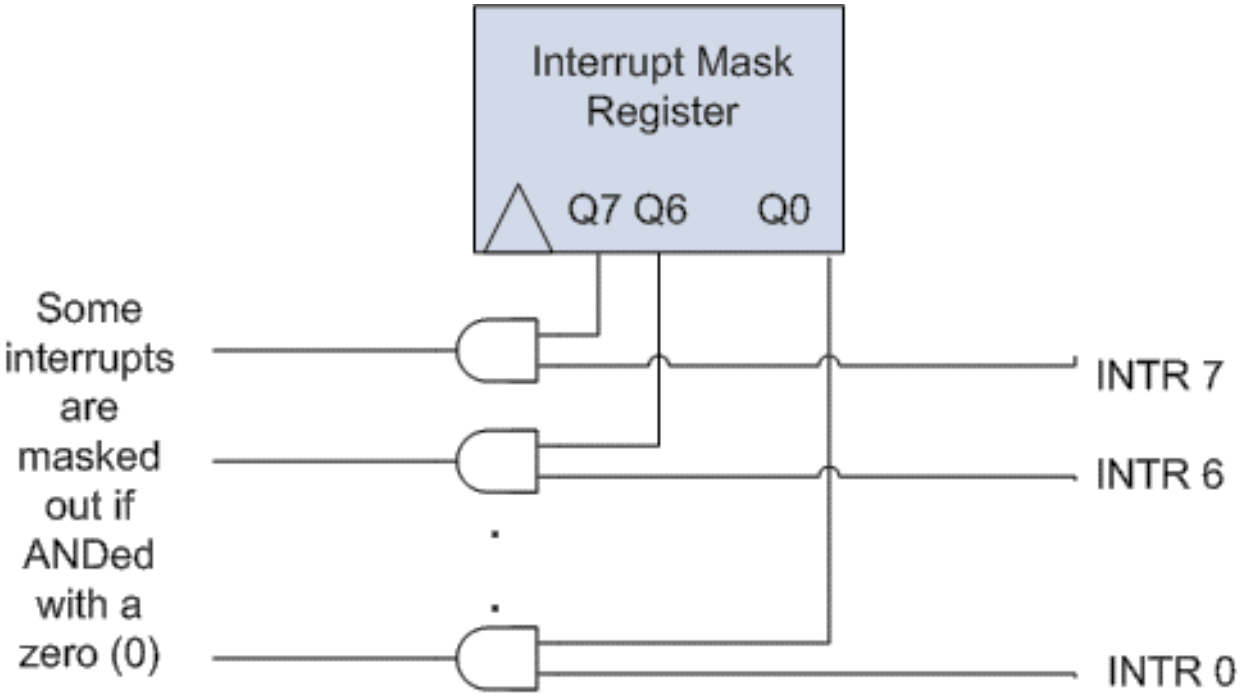
Execution of the interrupted program resumes after completion of execution of the interrupt-service routine. The concept of interrupts is useful in operating systems and in many control applications where processing of certain routines has to be accurately timed relative to external events. The latter type of application is generally referred to as *real-time processing*.

Interrupt Handling

The facilities in a computer should give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the CPU to suspend the execution of one program and start the execution of another. Because interrupts can *arrive at any time*, they may alter the sequence of events from that envisaged by the programmer. Hence, they should be carefully controlled. A fundamental facility found in all computers is the ability to enable and disable the occurrence of program interruptions as desired. We will now examine such facilities in some detail.

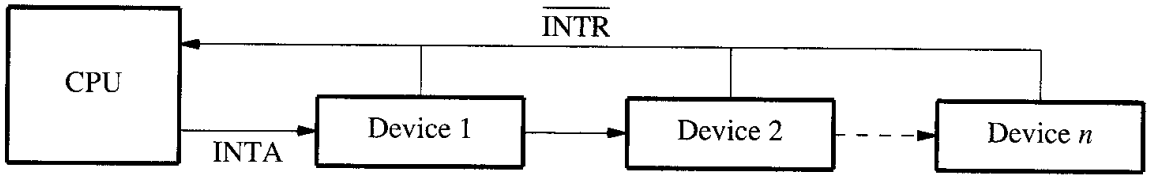
Enabling and Disabling Interrupts

There are many situations in which the CPU should ignore interrupt requests. For example, in the case of the Compute-Print program above, an interrupt request from the printer should be accepted only if there are output lines to be printed. After printing the last line of a set of ***n*** lines, interrupts should be disabled until another set becomes available for printing.

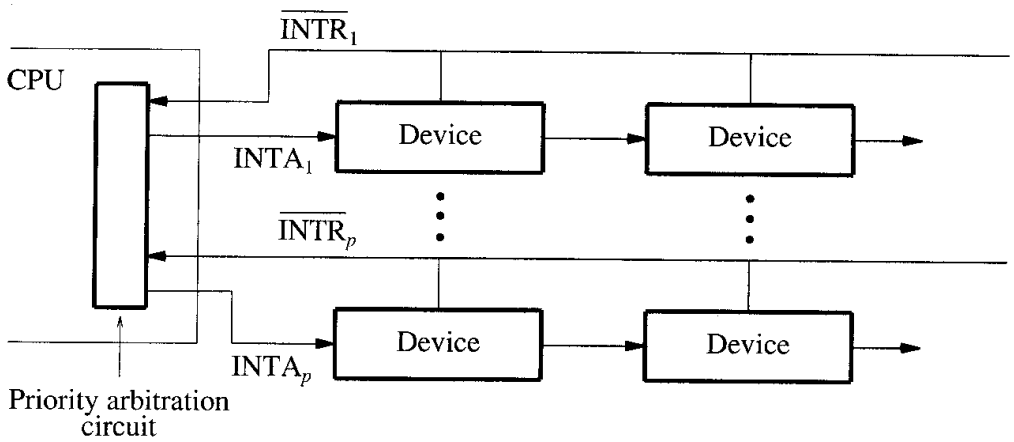


In another case, it may be necessary to guarantee that a particular sequence of instructions is executed to the end without interruption, because the interrupt-service-routine may change, or adversely affect, some of the data used by the instructions in question. For these reasons, some means for enabling and disabling interrupts must be available to the programmer. A simple way is to provide machine instructions, such as Interrupt-enable and Interrupt-disable, that perform these functions. For systems with multiple interrupt lines which are prioritized, an interrupt mask is provided. Entering "1"s and "0"s into the *Interrupt Mask Register* allows the programmer selectively to keep certain peripherals available while silencing others. Each bit in the interrupt mask is AND-ed with one of the interrupt lines before that line reaches the CPU. A zero in the interrupt mask will disable the interrupt with which it is partnered.

Let us consider in some detail the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the CPU has accepted its request. This means that the interrupt-request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. It is essential to ensure that this active request signal does not cause a second interruption during this period. An erroneous interpretation of a *single interrupt* as *multiple requests* would cause the system to enter an infinite loop from which it could not recover. Several mechanisms are available to solve this problem. Three simple possibilities here; other schemes that can handle more than one interrupting device will be presented later.



(a) Daisy chain



(b) Arrangement of priority groups

Note: 1 through 3 assume that there is just a single interrupt line, not a prioritized set. (a) above.

1. The first possibility is to have the CPU hardware ignore the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Thus, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, this will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. Again, the CPU must guarantee that execution of the Return-from-interrupt instruction is completed before further interruption can occur.

2. Another option, which was commonly encountered in practice, is to have the CPU automatically disable interrupts before starting the execution of the interrupt-service routine. That is, after saving the contents of the PC and the appropriate registers on the stack, the CPU automatically performs the equivalent of executing an Interrupt-disable instruction, if the system has just one interrupt line.

3. The third approach that can be used is to arrange the interrupt-handling circuit in the CPU such that it responds only to the leading or rising edge of the interrupt-request signal. Only one such transition will be seen by the CPU for every request generated by the device. Such interrupt-request lines are said to be edge-triggered.

Before proceeding to study more complex aspects and arrangements of interrupts, let us summarize the sequence of events involved in handling an interrupt request from a single device using just the one interrupt line: (a), above:

1. A program instruction enables interrupts in the CPU.
2. The device raises an interrupt request.
3. The CPU FINISHES THE CURRENT INSTRUCTION and then interrupts the program being executed at the time.
4. Interrupts are disabled.
5. The device is informed that its request has been recognized via **INTRA**, and in response, it deactivates the interrupt-request signal, **INTR**. (Handshaking signals).
6. The action requested by the interrupt is performed.
7. Interrupts are re-enabled.
8. Execution of the interrupted program is resumed after restoring the state of the CPU to that which it had when the running program was interrupted.

Handling Multiple Devices

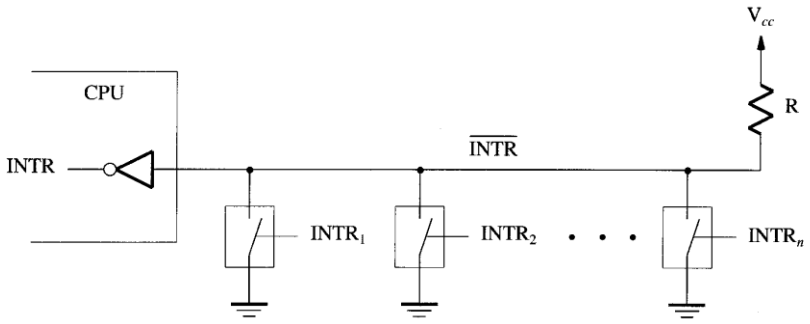
Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the CPU. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or all devices may request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the CPU recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt-service routines, how can the CPU obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the CPU while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

The means by which the above problems are resolved vary considerably from one machine to another. The approach taken in any machine is an important consideration in determining its suitability for a given application. We will now treat some of the more commonly used techniques.

Device Identification

Consider the case where an external device requests an interrupt by activating a single interrupt-request line that is common to all devices, say ***n*** devices.



An equivalent circuit for an open-collector bus used to implement a common interrupt-request line.

All ***n*** devices are attached to a common **INTR** line. When a device raises an interrupt by closing its switch, the interrupt-request line, **INTR**, is received via a line connected into the CPU.

When a request is received over the common interrupt-request line of the above figure, additional information is needed to identify the particular device that activated the line. The required information is available in the status registers of the devices. Hence, the interrupt-service routine should begin by surveying all the devices in some order.

The first device encountered with its Ready bit set is the device that should be serviced, and an appropriate subroutine should be called to provide the requested service.

This surveying scheme is very simple and easy to implement. Its main disadvantage is the time spent interrogating the status bits of all the devices that may not be requesting any service. An alternative approach is to use vectored interrupts, which we examine next.

Vectored Interrupts

In order to reduce the overhead involved in the surveying process, a device requesting an interrupt may identify itself directly to the CPU. Then, the CPU can immediately start executing the corresponding interrupt-service routine. The term ***vectored interrupts*** refers to all interrupt-handling schemes based on this approach.

A device requesting an interrupt may identify itself by sending a *device ID number* to the CPU over the bus. This technique enables identification of individual devices even if they share this single interrupt-request line. The *device ID number* supplied by the device may represent the starting address of the interrupt-service routine for that device, or, more often, it is used to locate and entry in a table which contains a *list* of the starting addresses of all ***I.S.Rs.***

In any given system, there is a limit to the number of devices which can be connected. 256 is a common figure.

So, this *device ID number* leads us to an entry in the vectored interrupt table, from where we get the starting address of the correct **I.S.R**. This constitutes a new value for the PC.

Some modifications to the hardware are required to support vectored interrupts. The key modification follows from the realization that the CPU may not respond immediately when it receives an interrupt request. The minimum delay in the CPU's response results from the requirement to ***complete execution of the current instruction***. Further delays may occur if interrupts happen to be disabled at the time the request is received. The necessary coordination can be achieved through the use of another control signal that may be termed interrupt acknowledge (**INTRA**). As soon as the CPU is ready to service the interrupt, it activates the **INTRA** line. This, in turn, can be made to cause the device interface to place the device ID number on the either the data lines of the bus and to turn off the **INTR** signal. **INTR and INTRA** are therefore a handshaking pair.

Interrupt Nesting

I suggested (in the case of a *single* interrupt line) that interrupts should be disabled during the execution of an interrupt-service routine, **I.S.R.** This ensures that an interrupt request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, **I.S.R**,, once started, always continues to completion before a second interrupt request is accepted by the CPU. Interrupt-service routines are typically short, and the delay they may cause in responding to a second request is acceptable for most ***simple*** computer systems.

For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation. Consider, for example, a computer that keeps track of the time of day using a real-time clock. This is a device that sends interrupt requests to the CPU at regular intervals. For each of these requests, the CPU executes a short interrupt-service routine **(I.S.R)** to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on. Proper operation requires that the delay in responding to an interrupt request from the real-time clock be small in comparison with the interval between two successive requests. To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept an interrupt request from the clock during the execution of an interrupt-service routine for another device.

This is an argument for *priorities* among interrupts, and therefore the existence of multiple **INTR** lines.

So, this example of the real-time clock suggests that I/O devices should be organized in a *priority structure* and we must acknowledge that this course has to consider the variety of computer systems which are in use *and not just focus on the machines that we use on a daily basis to the exclusion of small microprocessor systems and embedded systems for which the foregoing policies and circuits are still appropriate.*

Priorities and interrupts

However, returning to the commonly used individual laptops/desktops, we must implement a more complex system of interrupts involving an entire set of prioritized **INTR/INTRA** lines and an interrupt mask.

We imagine this: **An interrupt request from a high-priority device should be accepted while the CPU is servicing another request from a lower-priority device**.

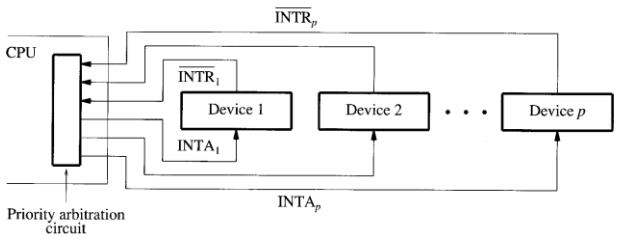
A multiple-level priority organization means that during execution of an interrupt-service routine, interrupt requests will be accepted from some devices but not from others, **depending upon the device's priority**. In order to facilitate implementation of this scheme, it is useful to assign a priority level to the CPU that can be changed under program control. The priority level of the CPU is the priority of the program that is currently being executed.

The CPU accepts interrupts only from devices that have priorities higher than its own.

At the time the execution of an interrupt-service routine for some device is started, the priority of the CPU should be raised to that of the device. This action disables interrupts from devices at the same level of priority or lower. However, interrupt requests from higher-priority devices will continue to be accepted.

The CPU priority is usually encoded in a few bits. 3 bits and you have 8 priority levels, 4 bits and there are 16.

The priority assigned to the CPU may be changed by a program instruction. Such instructions are usually privileged instructions, which are only executed by the operating system. Thus, a user program cannot accidentally or intentionally change the CPU priority and disrupt the system's operation.



Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

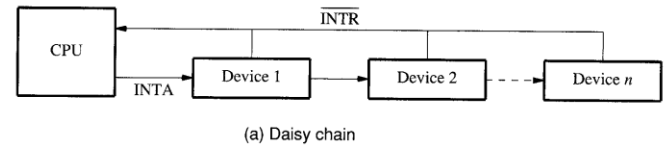
From the hardware point of view, a multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device. Such an arrangement is shown in this figure at left. Each of the interrupt request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the CPU. A request is accepted only if it has a higher priority level than that currently assigned to the CPU.

Simultaneous Requests

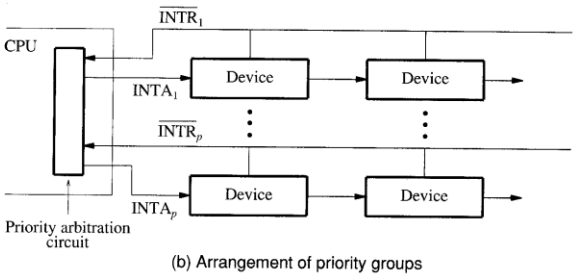
Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The CPU should have some means of arbitration by which only one request is serviced and the others are either delayed or ignored.

In the presence of a priority scheme such as that of the last figure, the solution to this problem is straightforward. The CPU simply accepts the request having the highest priority. However, if several devices share a particular interrupt-request line, some other mechanism must be employed to assign relative priority to these devices.

When surveying is used to identify the interrupting device, priority is automatically assigned by the order in which devices are surveyed. Therefore, no further treatment is required to accommodate simultaneous interrupt requests.



(a) Daisy chain



(b) Arrangement of priority groups

In the case of vectored interrupts, the priority of any device is usually determined by the way in which it is connected to the CPU. The scheme of figure part (a) has the advantage that it requires considerably fewer wires than the individual connections of the figure at left. The main advantage of this scheme is that it makes it possible for the CPU to accept interrupt requests from some devices, but not from others, and this is under programmer control via the **interrupt** mask, and depends upon their priorities.

The two schemes may be combined to produce the more general structure of figure part (b). This organization is used in many computer systems, including the 68000.

We should note that the general organization of figure part (b) makes it possible for a device to be ***connected to several priority levels***. At any given time, the device requests an interrupt at the priority level consistent with the urgency of the function being performed. This approach offers additional flexibility but requires more complex control circuitry in the device interface.

Controlling Device Requests

So far in our discussion of interrupts, we have assumed that any I/O device interface generates an interrupt request whenever it is ready for an I/O transfer, that is, whenever the Ready bit in its status register is equal to 1. It is important to ensure that interrupt requests are generated only by those I/O devices that are being used

by a given program. ***Idle devices must not be allowed to generate interrupt requests even though they may be ready to participate in I/O transfer operations***.

Hence, a facility is needed to enable and disable interrupts in the interface circuit of individual devices, in order to control whether the device is allowed to generate an interrupt request. Such a facility is usually provided in the form of an Interrupt-enable bit in the devices' interface circuit, which may be set or cleared by the CPU.

The Interrupt-enable bit may be a part of a control register, probably memory mapped, into which the CPU can write. When this bit is set, the interface circuit generates an interrupt request whenever its Ready bit is set. If it is a 0, the interface circuit will not generate an interrupt request even if the Ready bit is set.

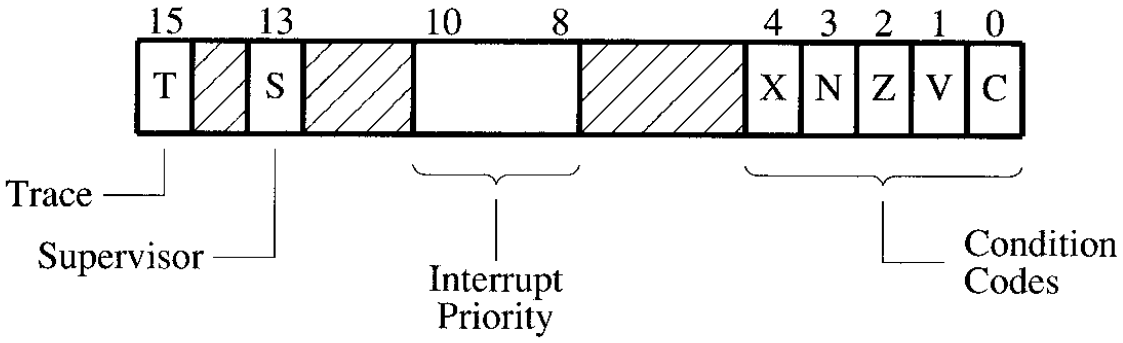
To summarize, there are two independent facilities that control interrupt requests. At the device end, an Interrupt-enable bit in a control register determines whether the device *is allowed* to generate an interrupt request.

At the CPU end, a priority structure and an ***interrupt mask*** determine whether a given interrupt request will be accepted.

We have discussed the organizational aspects of interrupts in general. Now lets describe the interrupt-handling mechanism of the 68000 chip as an example.

68000 Interrupt Structure

The 68000 has eight (8) interrupt priority levels. The priority at which the processor is running at any given time is encoded in three bits of the *processor status word*, which also contains the flags Level 0 is the lowest priority. The 68000 using an arrangement similar to that in the figure, where interrupt requests are assigned priorities in the range 1 to 7. A request is accepted only if its priority is higher than that of the processor, with one exception: An interrupt request at level seven is always accepted. It is called a non-maskable interrupt. When the CPU accepts an interrupt request, the priority level indicated in the Program Status Word (below) is automatically raised to that of the request, before the interrupt-service routine is executed. Thus, requests of equal or lower priority are disabled, except for level seven interrupts, which are always enabled. ***They are unmaskable and most likely associated with bus errors, memory errors or other hardware issues. A typical response (ISR) would do a memory dump so that the source of the hardware error might be traced.***



*In the above, "X" usually is equal to "C". It is sort of "unused". This register is called the **PSW,** the Program status word.*

The 68000 microprocessor uses vectored interrupts. When it accepts an interrupt request, it obtains the starting address of the interrupt-service routine from an *interrupt vector table* stored in the main memory. There are 256 interrupt vectors, numbered 0 to 255. Each line in this table consists of 32 bits that constitute the required starting address of the **ISR**.

When a device requests an interrupt, it may point to the interrupt-vector-table entry that should be used by sending an 8-bit vector number to the processor, in response to the interrupt acknowledge signal, **INTRA**. This would be the offset to the correct entry in the table. As an alternative, the 68000 also provides an auto-vector facility. Instead of sending a vector number, the device may activate a special bus control line to indicate that it wishes to use the auto-vector facility. In this case, the CPU chooses one of seven vectors provided for this purpose, based on the priority level of the interrupt request. These seven entries will probably use a shared **ISR**, but in any case, if invoked, the system likely has serious problems.

All in all, chip, system and implementation dependent.

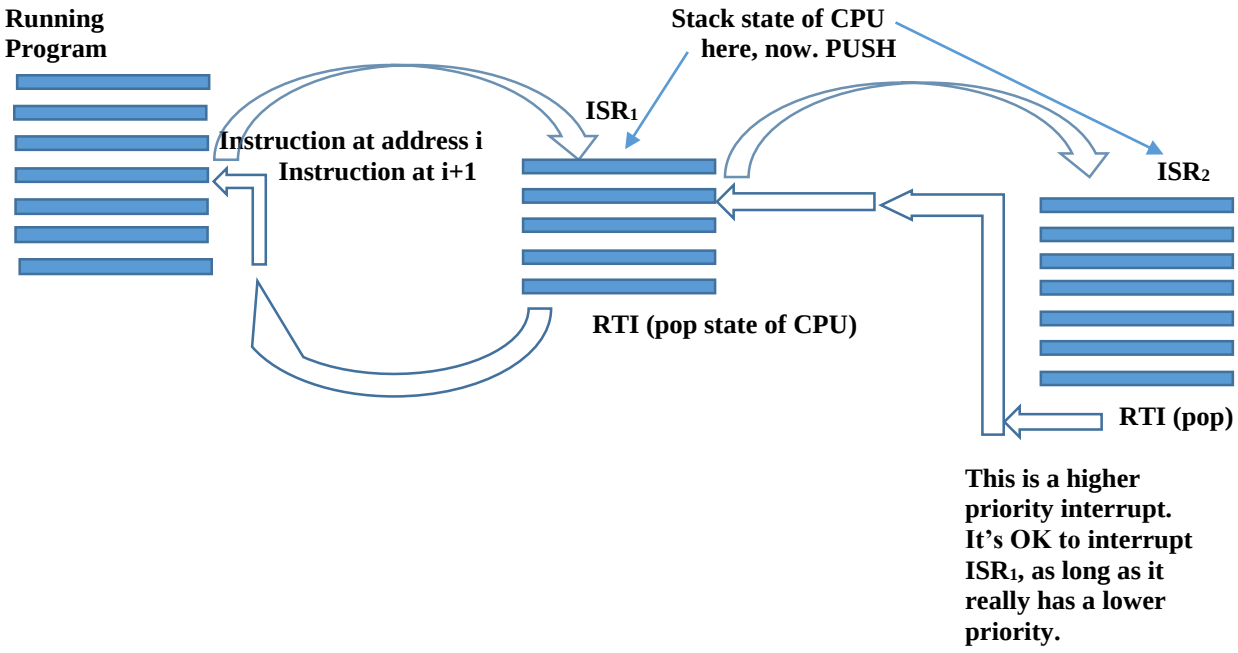
Endnote: What is the difference between memory mapped I/O and I/O ports?

Memory-mapped I/O (**MMIO**) and **port-mapped I/O** (**PMIO**) (which is also called ***isolated I/O***) are two complementary methods of performing input/output (I/O) between the CPU and peripheral devices in a computer system. An alternative approach is using dedicated I/O processors, commonly known as channels on mainframe computers, which execute their own instructions.

Nested Interrupts

Memory-mapped I/O uses the same VM address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register/memory. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation may be permanent or temporary; an example for the latter was the Commodore 64 that used bank switching between its I/O devices and regular memory.

Port-mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as the `in` and `out` instructions found on microprocessors based on the x86 and x86-64 architectures. Different forms of these two instructions can copy one, two or four bytes (`outb`, `outw` and `outl`, respectively) between the EAX (a *general* register) register or one of that register's subdivisions on the CPU and a specified I/O port which is assigned to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, **or an entire bus dedicated to I/O**. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.



Chains of **ISRs** can be tolerated as long as they are controlled. Only allowing a higher priority interrupt to add to the "chain", like the above, ensures a maximum "chain" length. Sizing the stack to accommodate the maximum chain length ensures that the system will never run out of stack space and this prevents crashes!