8.2 Programmed I/O

Programmed I/O takes place when an instruction in the program initiates the data transfer; for example, a programmer might write the instruction `MOVE.B Keyboard,D0` that reads a byte of data from the keyboard and puts it in D0. Some microprocessors have special instructions that are used only for I/O; for example, when a microprocessor executes an `OUT 123` operation, the contents of a data register are placed on the data bus. At the same time the number 123 is placed on the eight least-significant bits of the address bus and a pulse is generated on the system's I/O write line. Each of the I/O ports in such a system monitors the address lines. When an I/O interface sees its own address together with a read-port or a write-port signal, the interface acts on that signal and executes an I/O data transfer.

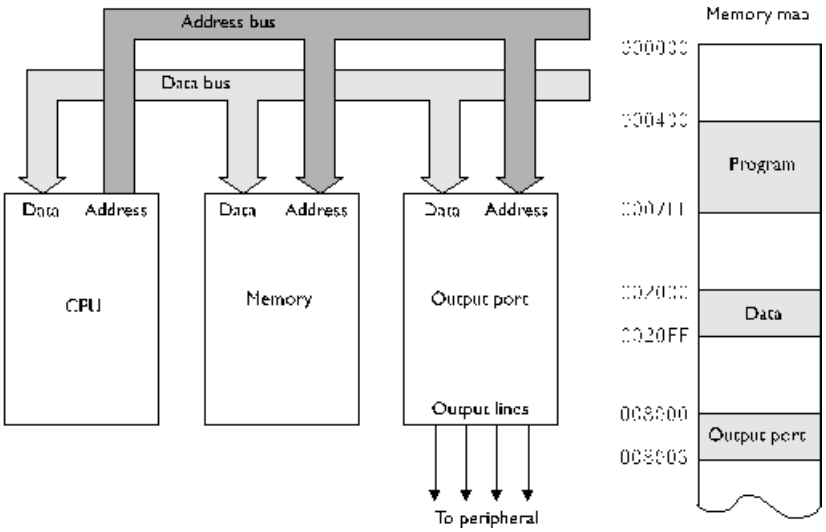
Memory-mapped I/O

Many microprocessors lack explicit I/O instructions like the `OUT <port>` we've just described. In fact, most microprocessors have no special input or output instructions whatsoever. If these microprocessors are to use programmed I/O, they have resort to *memory-mapped I/O* in which the processor treats interface ports as an extension to memory. Part of the CPU's normal memory space is dedicated to I/O operations and all I/O ports look exactly like normal memory locations.

Memory-mapped I/O ports are accessed by conventional memory reference instructions such as `MOVE D0,IO_PORT` (to output data) and `MOVE IO_PORT,D0` (to input data). A disadvantage of memory-mapped I/O is that part of the memory space available to programs and data is lost to the I/O system.

Figure 8.6 describes the organization of a memory-mapped I/O port and provides a memory map. An output port located at address $008000 is connected to a display device. Data is transmitted to the display by storing it in memory location $008000. As far as the processor is concerned, it's merely storing data in memory. The program in Table 8.1 sends 128 characters (starting at $002000) to the display.

Figure 8.6 Memory mapped I/O



The numbers in the right-hand column in table 8.1 give the time to execute each instruction in microseconds, assuming a clock rate of 8 MHz. To output the 128 characters takes approximately 128 x (8 + 8 + 8 + 10)/8 = 544 ms, which is a little over ½ thousandth of a second. Data is transferred at a rate of one character per 4¼ ms.

Table 8.1 A hypothetical example of a programmed output transfer

```
*  FOR i = 1 to 128
*     Move data from Table_i to output_port
*  ENDFOR
*                                                                    Cycles
PORT  EQU    $008000    Location of memory-mapped port
```

```
COUNT EQU    128        Size of block to be output
      ORG    $000400    Origin of program
*
      MOVE   #COUNT,D1  [D1] ¬ 128        Set up loop counter
      LEA    TABLE,A0   [A0] ¬ TABLE      A0 points to the table
      LEA    PORT,A1    [A1] ¬ Port1      A1 points to the port
*
LOOP  MOVE.B (A0)+,D0   [D0] ¬ [M([A0])]  Get data to be output            8
                        [A0] ¬ [A0] + 1
      MOVE.B D0,(A1)    [M([A1])] ¬ [D0]  Output the data                  8
      SUB    #1,D1      [D1] ¬ [D1] - 1   Decrement counter                8
      BNE    LOOP       Repeat until counter = 0                          10

      ORG    $002000    Origin for data area
TABLE DS.B   128        Reserve 128 bytes for the table of data
```

Although the program in Table 8.1 looks as if it should work, it's unsuited to almost all situations involving programmed output. Most peripherals connected to an output port are relatively slow devices and sending data to them at this rate would simply result in almost all the data being lost. As we've said, some interfaces are able to deal with short bursts of high-speed data because they store data in a buffer. They can't however, deal with a continuous stream of data at high speeds because the "waiting room" fills up and soon overflows.
A solution to the problem of dealing with such a mismatch in speed between computer and peripheral is found by asking the peripheral if it's ready to receive or to transmit data, and not sending data to it until it is ready to receive it. That is, we introduce a form of software handshaking procedure between the peripheral and the interface.

Almost all memory-mapped I/O ports occupy two or more memory locations. One location is reserved for the actual data to be input or output, and one holds a *status byte* associated with the port. For example, let $008000 be the location of the port to which data is sent and let $008002 be the location of the status byte. Suppose that bit 0 of the status byte is a 1 if the port is ready for data and a 0 if it is busy. The fragment of program in Table 8.2 implements memory-mapped output at a rate determined by the peripheral. The comments at the beginning of the program describe the data transfer in pseudocode.

Table 8.2 Using the polling loop to control the flow of data

```
*  FOR i = 1 TO 128
*          REPEAT
*             Read Port_status_byte
*          UNTIL Port_not_busy
*          Move data from Table_i to output_port
*  ENDFOR
*
1.  PORTDATA  EQU    $008000       Location of memory-mapped port
2.  PORTSTAT  EQU    $008002       Location of port's status byte
3.  COUNT     EQU    128           Size of block to be output
4.            ORG    $000400       Origin of program
5.            MOVE   #COUNT,D1     Set up character counter in D1
6.            LEA    TABLE,A0      A0 points to table in memory
7.            LEA    PORTDATA,A1   A1 points to data port
8.            LEA    PORTSTAT,A2   A2 points to port status byte
9.  LOOP      MOVE.B (A0)+,D0      Get a byte from the table
10. WAIT      MOVE.B (A2),D2         REPEAT Read the port's status
11.           AND.B  #1,D2             Mask all but lsb of status
12.           BEQ    WAIT            Until port ready
13.           MOVE.B D0,(A1)       Store data in peripheral
14.           SUB    #1,D1         Decrement loop counter
15.           BNE    LOOP          Repeat until COUNT = 0
16. *
17.           ORG    $002000       Start of data area
18. TABLE     DS.B   128           Reserve 128 bytes of data
```

The program in Table 8.2 is identical to the previous example in table 8.1 except for lines 8 to 12 inclusive. In line 8 an address register, A2, is used to point to the status byte of the interface at address $008002. In line 10 the status byte of the interface is read into D2 and masked down to the least-significant bit (by the action of `AND.B #1,D2` in line 11). If the least-significant bit of the status byte is zero, a branch back to line 10 is made by the instruction in line 12. When the interface becomes free (as indicated by the least-significant bit of the status byte being 1), the branch to `WAIT` is not taken and the program continues exactly as in Table 8.1.

Lines 10, 11, and 12 constitute a *polling loop*, in which the output device is continually polled (questioned) until it indicates it is free, allowing the program to continue. A typical slow printer operates at 30 characters/second, or approximately 1 character per 33,000 ms. Because the polling loop takes about 3 ms, the loop is executed 11,000 times per character.

Operating a computer in a polled input/output mode is grossly inefficient because so much of the computer's time is wasted waiting for the port to become free. If the microcomputer has nothing better to do while it is waiting for a peripheral to become free (i.e., not busy) polled I/O is perfectly acceptable. Many of the first-generation of personal computers were operated in this way. However, a more powerful computer working in a multiprogramming environment can attend to another task program during the time the I/O port is busy. In this case a better I/O strategy is to ignore the peripheral until it is ready for a data transfer and then let the peripheral ask the CPU for attention. Such a strategy is called *interrupt-driven I/O*. Note that all the I/O strategies we are describing use memory-mapped I/O.

By the way, if you are designing a computer with memory-mapped I/O and a memory cache, you have to tell the cache controller not to cache the port's status register. If you don't do this, the cache memory would read the status once, cache it, and then return the cached value on successive accesses to the status. Even if the status register in the peripheral changes, the old value in the cache is frozen.

8.3 Interrupt-driven I/O

A computer executes instructions sequentially unless a jump or a branch is made. There is, however, an important exception to this rule called an *interrupt*. An interrupt is an *event* that forces the CPU to modify its sequence of actions. This "event" may be an external signal from a peripheral (i.e., a *hardware interrupt*) or an internally generated call to the operating system (i.e., a *software interrupt*). Today, the term *exception* is often used to describe all these events.

Most microprocessors have an active-low *interrupt request* input called IRQ that can be asserted by a peripheral to request attention. Note that the word *request* carries with it the implication that the interrupt request may or may not be granted. Figure 8.7 illustrates the organization of a system with a simple interrupt-driven I/O mechanism.

Figure 8.7 Interrupt organization

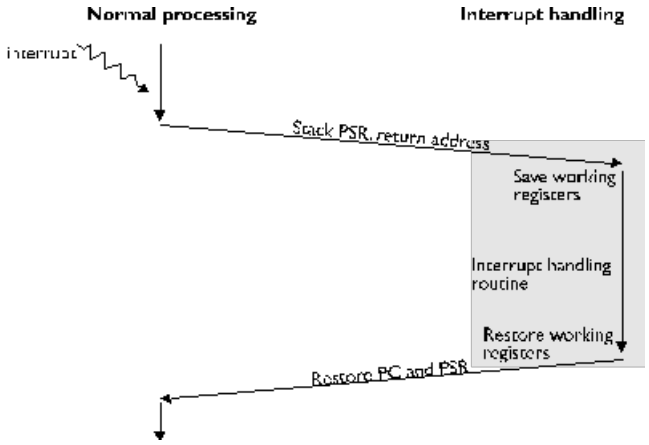
In Figure 8.7 an interrupt request line runs from the CPU to all the peripherals. Each peripheral capable of generating an interrupt is connected to this IRQ line. A peripheral asserts its IRQ output when it requires attention. Figure 8.7 demonstrates that few components are needed to implement interrupt-driven I/O in a microprocessor system. This system is analogous to the emergency handle in a train. When the handle is pulled in one of the carriages, the driver knows that a problem has arisen but doesn't yet know who pulled the handle. Similarly, the CPU doesn't know which peripheral caused the interrupt or why.

When the CPU detects that its IRQ input has gone active-low, the following (simplified) sequence of events takes place.

- The CPU finishes its current instruction. Most microprocessors cannot be stopped in mid-instruction, because individual machine code instructions are *indivisible* and must always be executed to completion.
- The contents of the program counter and the condition code register are pushed onto the stack. The *processor status word*, PSW, must be saved because the interrupt routine will almost certainly modify the condition code bits.
- Further interrupts are disabled to avoid an interrupt being interrupted (we will elaborate on this partially true statement later).
- The CPU deals with the cause of the interrupt by executing a program called an *interrupt handler*.
- The CPU executes a *return from interrupt* instruction at the end of the interrupt handler. Executing this instruction pulls the PC and PSW off the stack and execution then continues normally—as if the interrupt had never happened.

Figure 8.8 illustrates the sequence of actions taking place when an interrupt occurs. Note that (in a 68000 system) the processor status word consists of the system byte plus the condition code register, CCR. The system byte is used by the operating system and interrupt processing mechanism.

Figure 8.8 Interrupt sequence



Interrupt-driven I/O requires a more complex program than programmed I/O because the information transfer takes place not when the programmer wants or expects it, but when the data is available. The software required to implement interrupt-driven I/O is frequently part of the operating system. A fragment of a hypothetical interrupt-driven output routine in 68000 assembly language is provided in Table 8.3. Each time the interrupt handling routine is called, data is obtained from a buffer and passed to the memory-mapped output port at $008000. In a practical system some check would be needed to test for the end of the buffer.

Table 8.3 A simple interrupt handler

```
*  Pick up pointer to next free entry in the table (buffer)
*  Read a byte from the table and transmit it to the interface
*  Move the pointer to the next entry in the table
*  and save the pointer in memory
*  Return from interrupt
*
OUTPUT  EQU    $008000         Location of memory-mapped output port
        ORG    $000400         Start of the program fragment
*
        LEA    POINTER,A0      Load A0 with the pointer to the buffer
        MOVE.B (A0)+,D0        Read a character from the buffer
        MOVE.B D0,OUTPUT       Send this character to the output port
        MOVE.L A0,POINTER      Save the updated pointer
        RTE                    Return from interrupt

        ORG    $002000         Data origin
BUFFER  DS.B   1024            Reserve 1024 bytes for the table
POINTER DS.L   1               Reserve a longword for the pointer
```

Because the processor executes this code only when a peripheral requests an I/O transaction, interrupt-driven I/O is very much more efficient than the polled-I/O we described earlier.

Although the basic idea of interrupts is common to most computers, there are considerable variations in the precise nature of the interrupt-handling structure from computer to computer. We are now going to look at how the 68000 deals with interrupts because this microprocessor has a particularly comprehensive interrupt handling facility.

8.3.1 Prioritized Interrupts

Computer interrupts are almost exactly analogous to interrupts in everyday life. Suppose two students interrupt me when I'm lecturing—one with a question and the other because they feel unwell. I will respond to the more urgent of the two requests. Once I've dealt with the student who's unwell, I answer the other student's question and then continue my teaching. Computers behave in the same way.

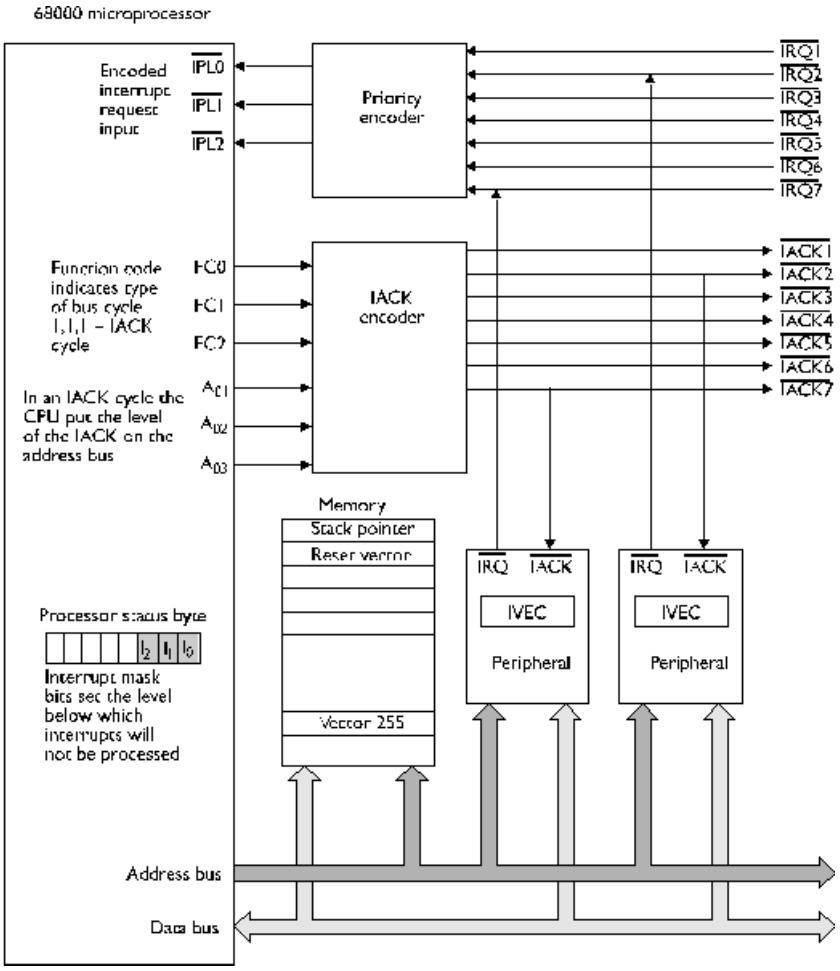
Most computers have more than one interrupt request input. Some interrupt request pins are connected to peripherals requiring immediate attention (e.g., a disk drive), whereas others are connected to peripherals requiring less urgent attention (e.g., a keyboard). For the sake of accuracy, we should point out that the processor's interrupt request input is connected to the peripheral's interface, rather than the peripheral itself. If the disk drive is not attended to (i.e., serviced) when its data is available, the data will soon be lost. Data from the disk is lost because it is replaced by new data. In such circumstances, it is reasonable to assign a priority to each of the interrupt request pins.

The 68000 supports seven interrupt request inputs from IRQ7*, the most important to IRQ1*, the least important. Suppose an interrupt is caused by the assertion of IRQ3* and no other interrupts are pending. The interrupt on IRQ3* will be serviced. If an interrupt at a

level higher than IRQ3* occurs, it will be serviced before the level 3 interrupt service routine is completed. However, interrupts generated by IRQ1* or IRQ2* will be stored pending the completion of IRQ3*'s service routine.

The 68000 does not have seven explicit IRQ1* to IRQ7* interrupt request inputs (simply because such an arrangement would require seven precious pins). Instead, the 68000 has a three-bit encoded interrupt request input, IPL0* to IPL2*. The 3-bit value on IPL0* to IPL2* reflects the current level of interrupt request from 0 (i.e., no interrupt request) to 7 (the highest level corresponding to IRQ7*). Figure 8.9 illustrates some of the elements involved in the 68000's interrupt handling structure. A *priority encoder* chip is required to convert an interrupt request on IRQ1* to IRQ7* into a three-bit code in IPL0* to IPL2*. The priority encoder automatically prioritizes interrupt requests and its output reflects the highest interrupt request level asserted.

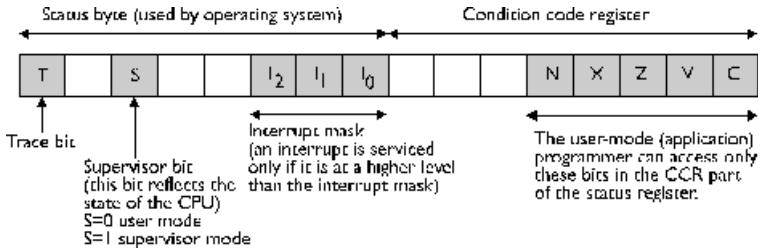
Figure 8.9 The 68000's interrupt structure



The 68000 doesn't automatically service an interrupt request. A *processor status byte* in the CPU in figure 8.9 controls the way in which the 68000 responds to an interrupt. Figure 8.10 describes the *status byte* in more detail. The three-bit *interrupt mask field* in the processor status byte, I_2, I_1, I_0, determines how the 68000 responds to an interrupt. The current interrupt request is serviced if its level is greater than that of the interrupt mask; otherwise the request is ignored. For example, if the interrupt mask has a current value of 4, only interrupt requests on IRQ5* to IRQ7* will be serviced.

When the 68000 services an interrupt, the interrupt mask bits are reset to make them equal to the level of the interrupt currently being serviced. For example, if the interrupt mask bits was set to 2 and an interrupt occurred at level IRQ5*, the mask bits would be set to 5. Consequently, the 68000 can now be re-interrupted only by interrupt levels 6 and 7. After the interrupt has been serviced, the old value of the processor status byte saved on the stack, and therefore the interrupt mask bits, are restored to their original level.

Figure 8.10 The 68000's status word



Non-Maskable Interrupts

Microprocessors sometimes have a special interrupt request input called a *non-maskable interrupt request*. The term *non-maskable* means that the interrupt cannot be turned off (i.e., delayed or suspended) and must be serviced immediately. Non-maskable interrupts are found in two particular applications. The first includes circumstances in which the interrupt is caused by a critical event that must not be missed. A typical event is an interruption of the power supply. When this happens the system still functions for a few milliseconds on energy stored in capacitors (energy storage devices found in all power supplies). A non-maskable interrupt generated at the first sign of a power loss is used to shut down the computer in an orderly fashion so that it can be re-started later with little loss of data and no corruption (accidental over-writing) of disk files.

A second application of non-maskable interrupts is in real-time systems. Suppose that the temperature and pressure at various points must be measured periodically in a chemical process. If these points aren't polled on a programmed basis, a stream of regularly spaced non-maskable interrupts will do the trick. At each interrupt, the contents of a counter register are updated and, if a suitable span of time has elapsed, the required readings are taken.

The 68000 reserves its level 7 interrupt (IRQ7*) as a non-maskable interrupt, because an interrupt on IRQ7* is *always* serviced by the 68000. If a level 7 interrupt is currently being serviced by the 68000, a further active transition on IRQ7* (i.e., a high-to-low edge) results in the 68000 servicing the new level 7 interrupt.

8.3.2 Vectored Interrupts

Remember that all interfaces capable of generating an interrupt are connected to a single interrupt request line (but forget for a moment the 68000's 7-level interrupt request system). When an interrupt occurs, the microprocessor reads the contents of a certain memory location containing a pointer to the interrupt hander, and jumps to the address stored in this location. Which location does the processor read to get the interrupt vector (i.e., pointer)? The answer depends on the actual processor and is *design dependent*. Some designers put the interrupt vector at the top end of memory and some put it at the bottom end.

A predetermined fixed address holding a pointer to an interrupt handling routine has several disadvantages. In particular, it's not possible to distinguish between interrupts originating from one of a several peripherals. Before we look at how the 68000 solves the problem of multiple interrupt sources, it's instructive to consider how first-generation microprocessors performed the task of isolating the cause of an interrupt request.

Following the detection and acceptance of an interrupt, first-generation microprocessors resorted to a simple method of locating the appropriate interrupt-handling routine. These processors used software to ask each of the possible interrupters, in turn, whether they were responsible for the interrupt. This operation is called *polling* (the same mechanism used for programmed I/O).

Figure 8.11 shows the structure of a typical memory-mapped I/O port with a data port at address $8000 and a status byte at location $8002. We have defined three bits in the status byte

- RDY (ready) indicates that the port is ready to take part in a data transaction
- IRQ indicates that the port has generated an interrupt
- ERR indicates that an error has occurred (i.e., the input or output is unreliable).

A purely polled I/O system simply tests the RDY bit of a peripheral until it is ready to take part in an I/O transaction. A system with interrupt-driven I/O and device polling waits for an interrupt and then reads the IRQ bit in the status register of each peripheral. This technique is fairly efficient as long as there are few devices capable of generating an interrupt.

Figure 8.11

Because the programmer chooses the order in which the interfaces are polled following an interrupt, a limited measure of prioritization is built into the polling process. However, a well-known law of the universe states that when searching through a pile of magazines for a particular copy, the desired issue is always at the opposite end to the point at which the search was started.

Incidentally, beginning the search at the other end of the pile doesn't seem to defeat this law. Likewise, the device that generated the interrupt is the last device to be polled. Consequently, a system with polled interrupts could lead to the situation in which a certain device requests service but never gets it. We next demonstrate how some processors allow the peripheral that requested attention to identify itself by means of a mechanism called the *vectored interrupt*.

The Vectored Interrupt

In a system with *vectored interrupts* the interface itself identifies its associated interrupt-handling routine, thereby removing the need for interrupt polling. Let's look at how the 68000 implements vectored interrupts. Whenever the 68000 detects an interrupt, the 68000 acknowledges it by transmitting an *interrupt acknowledge* (called IACK) message to all the interfaces that might have originated the interrupt.

The 68000 doesn't have an explicit "IACK" pin. Instead, it uses its three *function code outputs*, FC0, FC1, FC2, to inform peripherals that it's acknowledging an interrupt (see Figure 8.9). These three function code outputs tell external devices (memory and interfaces) what the 68000's doing. For example, the function code tells the system whether the 68000 is reading an instruction or an operand from memory. The special function code 1,1,1 indicates an interrupt acknowledge.

Because the 68000 has seven levels of interrupt request, it's necessary to acknowledge only the appropriate level of interrupt. It would be unfair if a level 2 and a level 6 interrupt occurred nearly simultaneously and the interface requesting a level 2 interrupt thought that its interrupt was about to be serviced. The 68000 indicates which level of interrupt it's acknowledging by providing the level of the interrupt being serviced on the three least-significant bits of its address bus (A_{01} to A_{03}). External logic detects FC0, FC1, FC2 = 1,1,1 and uses A_{01} to A_{03} to generate seven interrupt acknowledge signals IACK0* to IACK7*.

After issuing an interrupt request, the interface waits for an acknowledgement on its IACK input. When the interface detects IACK asserted, it puts out an *interrupt vector number* on data lines d_{00} to d_{07}. That is, the interface responds with a number ranging from 0 to 255. When the 68000 receives this interrupt vector number, it multiplies it by four to get an entry into the 68000's *interrupt vector table*; for example, if an interface responds to an IACK cycle with a vector number of 100, the CPU multiplies it by 4 to get 400. In the next step, the 68000 reads the contents of memory location 400 to get a pointer to the location of the interrupt handing routine for the interface that initiated the interrupt. This pointer is loaded into the 68000's program counter to start interrupt processing.

Because an interface can supply one of 256 possible vector numbers, it's theoretically possible to support 256 unique interrupt-handling routines for 256 different interfaces. We say theoretically, because it's unusual for 68000 systems to dedicate all 256 vector numbers to interrupt handling. In fact, the 68000 itself uses vector numbers 0 to 63 for purposes other than handling hardware interrupts (these vectors are reserved for other types of exception).

Why does the 68000 multiply the vector number by four? The answer to this is easy: the interrupt vector loaded into the 68000's PC is a 32-bit value that occupies four bytes in memory. Therefore, each vector number is associated with a four-byte (i.e., longword) block of memory. The interrupt vector table itself takes up 4 x 256 = 1,024 bytes of memory. Figure 8.12 illustrates the way in which the 68000 responds to a level 6 vectored interrupt.

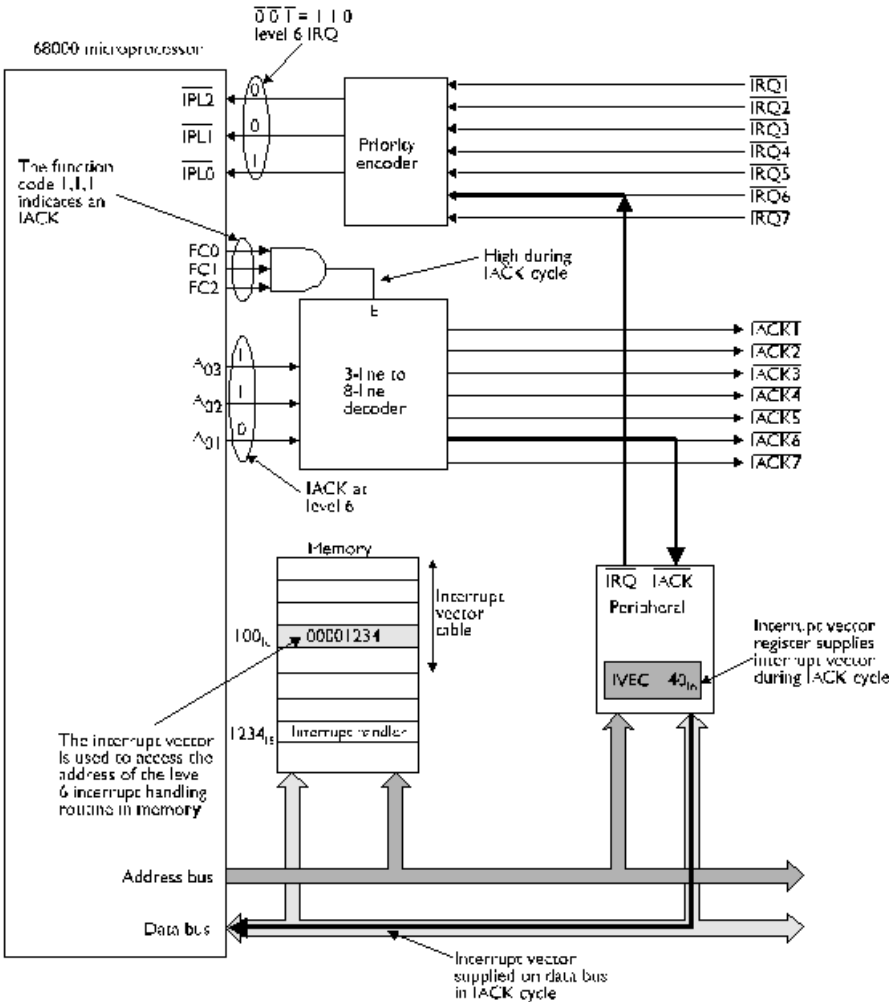
Daisy-Chaining

The scheme for vectored interrupts we've just described seems to have an important flaw. Although there are 256 possible interrupt vector numbers, the 68000 supports only seven levels of interrupt. Why then do we need to cater for so many interrupt handlers? In the first case, it is not necessary for all interfaces to be active at the same time. We could envisage a scheme with say 20 interfaces with 20 interrupt handlers. It is perfectly possible to program the interfaces so that only 7 of them can issue an interrupt at any instant.

A simple means of providing an effectively infinite number of levels of interrupt request is provided by a technique called *daisy-chaining* in which peripherals are linked together in a line. When the CPU acknowledges an interrupt, the message is sent to the first peripheral in the daisy chain. It this peripheral, doesn't require attention, it passes the interrupt acknowledgment down the line to the next peripheral.

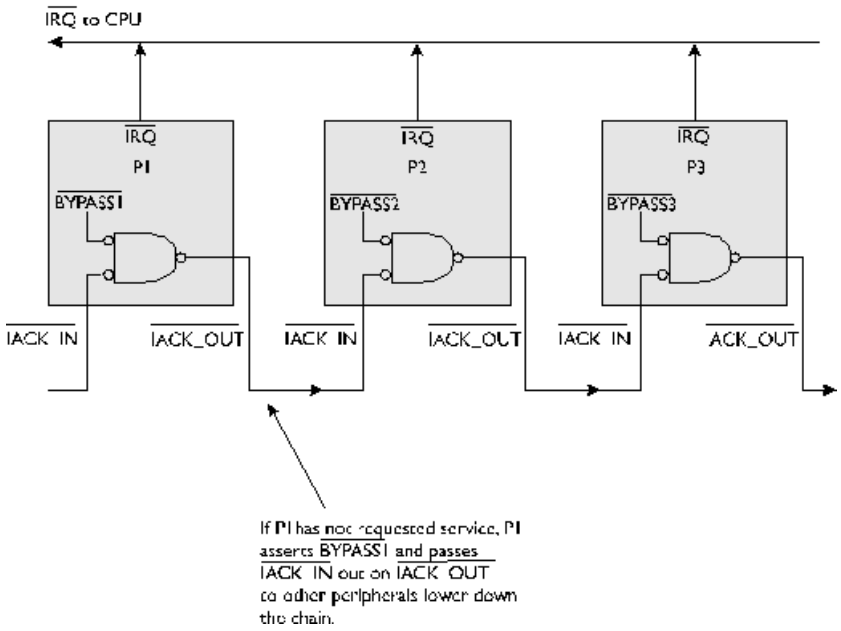
Figure 8.13 shows how interrupt requesters at a given priority level can be prioritized by *daisy chaining*. Each peripheral has an IACK_IN* input on its left and an IACK_OUT* output on its right. The IACK_OUT* pin of each peripheral is wired to the IACK_IN* pin of the peripheral on its right. Thus, the IACK_OUT*/IACK_IN* pins constitute a daisy chain. Suppose an interrupt request at level 6 is issued and acknowledged by the 68000. The interface at the left-hand side of the daisy chain closest to the 68000 receives the IACK* signal first from the CPU. If this interface generated the interrupt, it responds with an interrupt vector. If the interface did not request service, it passes the IACK* signal to the device on its right. That is, IACK_IN* is passed out on IACK_OUT*. The IACK* signal ripples down the daisy chain until a device responds with an interrupt vector.

Figure 8.12 Responding to a level 6 vectored interrupt



Daisy-chaining interfaces permits an unlimited number of interfaces to share the same level of interrupt and each interface to have its own interrupt vector number. Individual interfaces are prioritized by their position with respect to the CPU. The closer to the CPU an interface is, the more chance it has of having its interrupt request serviced in the event of multiple interrupt requests at this level.

Figure 8.13



Interrupts Considered Harmful

Each aspect of life can be divided into one of two categories: a good thing or a bad thing. Some computer scientists are firmly convinced that interrupts are a bad thing. A single interface generating the occasional interrupt causes few headaches. But imagine a system with many interfaces, all generating their interrupts asynchronously (i.e., at random). The entire system no longer behaves in a deterministic way but becomes stochastic (non-deterministic) and is best described by the mathematics of random processes. This system is analogous to a large group of people in a bar—there is always someone who never gets served.

Some safety-critical systems rely on polling to determine whether I/O devices are ready to take part in I/O transactions rather than interrupts. The programmer has control of all polled I/O operations and can choose how and when peripherals are polled.

8.4 Direct Memory Access

A computer computes, so why should it become involved with input/output activity that is not central to the processor's activity? The third I/O strategy called *direct memory access*, DMA, moves data between a peripheral and the CPU's memory without the direct intervention of the CPU itself. DMA provides the fastest possible means of transferring data between an interface and memory, as it carries no CPU overhead and leaves the CPU free to do useful work. As in most walks of life, if something is worth having, it's expensive. DMA is no exception to this rule, because it is quite complex to implement and requires a relatively large amount of hardware. Figure 8.14 illustrates the operation of a system with DMA.

Figure 8.14 Input/Output by means of DMA

DMA works by grabbing the data and address buses from the CPU and using them to transfer data directly between the peripheral and memory. During normal operation of the computer in figure 8.14, bus switch 1 is closed, and bus switches 2 and 3 are open. The CPU controls the buses, providing an address on the address bus and reading data from memory or writing data to memory via the data bus.

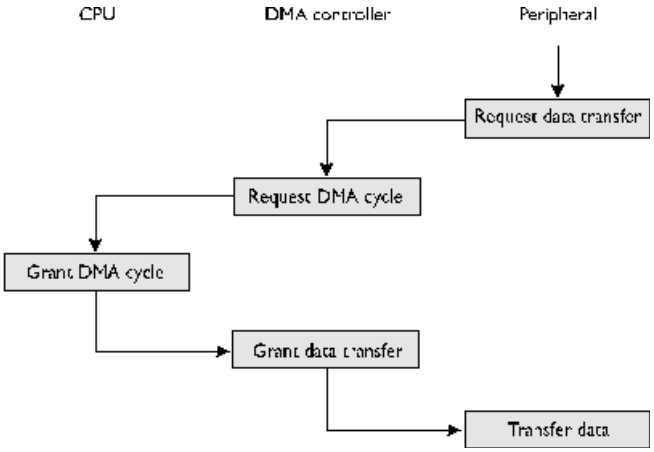
When a peripheral wishes to take part in an I/O transaction it asserts the TransferRequest input of the DMA controller (DMAC). In turn, the DMA controller asserts DMArequest to request control of the buses from the CPU; that is the CPU is taken off-line. When the CPU returns DMAgrant to the DMAC, a DMA transfer takes place. Bus switch 1 is opened and switches 2 and 3 closed. The DMAC provides an address to the address bus and hence to the memory. At the same time, the DMAC provides a TransferGrant signal to the peripheral that is then able to write to, or read from, the memory directly. When the DMA operation has been completed, the DMAC hands back control of the bus to the CPU.

A real DMA controller is a very complex device. It has several internal registers, at least one to hold the address of the next memory location to access and one to hold the number of words to be transferred. Many DMACs are able to handle with several interfaces, which means that their registers must be duplicated. Each interface is referred to as a *channel* and typical single-chip DMA controllers handle up to four channels (i.e., peripherals) simultaneously.

Figure 8.15 provides a protocol flowchart for the sequence of operations taking place during a DMA operation. This figure shows the sequence of events that take place in the form of a series of transactions between the peripheral, DMAC, and the CPU.

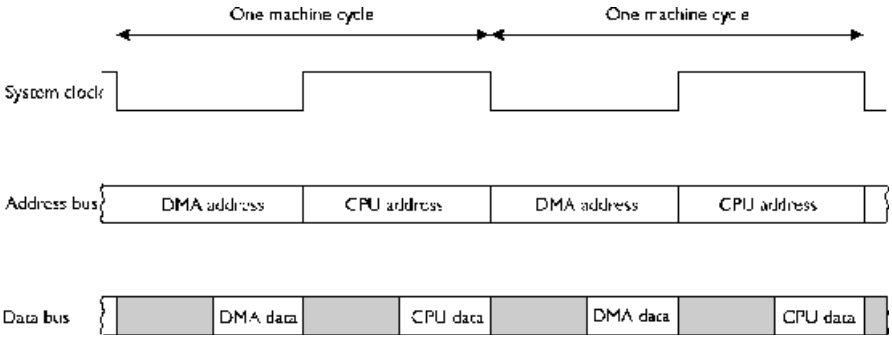
DMA operates in one of two modes: *burst mode* or *cycle stealing*. In the burst mode the DMA controller seizes the system bus for the duration of the data transfer operation (or at least for the transfer of a large number of words). Burst mode DMA allows data to be moved into memory as fast as the weakest link in the chain memory/bus/interface permits. The CPU is effectively halted in the burst mode because it cannot use its data and address buses.

Figure 8.15 Protocol flowchart for a DMA operation



In the cycle steal mode described by figure 8.16, DMA operations are interleaved with the computer's normal memory accesses. As the computer does not require access to the system buses for 100% of the time, DMA can take place when they are free. This free time occurs while the CPU is busy generating an address ready for a memory read or write cycle.

Figure 8.16 DMA by cycle-stealing



When the system clock is low, the CPU doesn't need the use of the buses, so the DMAC grabs them and carries out a data transfer. When the clock goes high the CPU carries out its normal memory access cycle. DMA by cycle stealing is said to be *transparent* because the computer is oblivious of it. That is, the transfer is invisible to the computer and no processing time is lost. A DMA operation is initiated by the CPU writing a start address and the number of words to be transferred into the DMAC's registers. When the DMA operation has been completed, the DMAC generates an interrupt, indicating to the CPU that the data transfer is over, and that a new one may be initiated or results of the current transfer made use of.

In systems with a cache memory, DMA can take place in parallel with normal CPU activity; that is, the CPU can access data and code that's been cached while the I/O interface is copying data between a peripheral and the main memory.

8.5 Parallel and Serial Interfaces

Having described how I/O transactions can be programmed, interrupt-driven, or use DMA, we now look at two real integrated circuits that implement an interface between the CPU and the outside world. These devices look like a block of memory locations to the CPU and implement the protocol required to communicate with the external system. Although we've decided to describe two actual devices, the general principles apply to virtually all interface devices. Readers not interested in the fine details of I/O systems may skip this section.

The first interface to be described is the peripheral interface adapter that transfers data between an external system and a processor via an 8-bit highway, and the second interface is the asynchronous communications adapter that transfers data on a single-bit serial highway. Both interfaces are basic first-generation circuits intended for applications involving eight-bit microprocessors. We have selected these particular devices (i.e., 6820 PIA and 6850 ACIA) because they illustrate the fundamental principles and are not as complex as the current crop of interfaces optimized for high-performance 16/32-bit microprocessors. Other interfaces found in microprocessor systems include: Ethernet controllers, disk controllers, and video controllers.

8.5.2 The Serial Interface

The serial interface transfers data into and out of the CPU a bit at a time along a single wire; that is, the 8-bit value 10110001_2 would be sent in the form of eight (or more) pulses one after the other. Serial data transfer is much slower than the parallel data transfer offered by a PIA, but is inexpensive because it requires only a single connection between the serial interface and the external world (apart from a ground-return). Serial data transmission is used by data transmission systems that operate over distances greater than a few meters and Chapter 11 will have much more to say on the subject of data transmission. Here we're more interested in the *asynchronous communications adapter* (ACIA) interface that connects a CPU to a serial data link.

We are not concerned with fine details of the ACIA's internal operation, but rather in what it does and how it is used to transmit and receive serial data. When discussing serial transmission we often use the term *character* to refer to a unit of data rather than byte, because many transmission systems are designed to transmit information in the form of ISO/ASCII-encoded characters.

Figure 8.24 demonstrates how a 7-bit character is transmitted bit by bit *asynchronously* (there are other ways of implementing serial transmission systems). During a period in which no data is being transmitted from an ACIA, the serial output is at a high level, which is called the *mark* condition. When a character is to be transmitted, the ACIA's serial output is put in a low state (a mark-to-space transition) for a period of one bit time. The bit time is the reciprocal of the rate at which successive serial bits are transmitted and is measured in *Baud*. In the case of a two-level binary signal, the Baud corresponds to bits/sec. The initial bit is called the *start bit* and tells the receiver that a stream of bits, representing a character, is about to be received. If data is transmitted at 9,600 Baud, each bit period is 1/9,600 = 0.1042 ms.

Figure 8.24 Format of an asynchronous serial character

During the next seven time slots (each of the same duration as the start bit) the output of the ACIA depends on the value of the character being transmitted. The character is transmitted bit-by-bit. This data format is called *non-return to zero* (NRZ) because the output doesn't go to zero between individual bits. After the character has been transmitted, a further two bits (a *parity bit* and a *stop bit*) are appended to the end of the character.

At the receiver, a new parity bit is generated locally from the incoming data and then compared with the received parity bit. If the received and locally generated parity bits differ, an error in transmission is assumed to have occurred. A simple parity bit cannot correct an error once it has occurred, nor detect a pair of errors in a character. Not all serial data transmission systems employ a parity bit as an error detector.

The stop bit (or optionally two stops bits) indicates the end of the character. Following the reception of the stop bit(s), the transmitter output is once more in its mark state and is ready to send the next character. The character is composed of ten bits but contains only seven bits of useful information.

The key to asynchronous data transmission is that once the receiver has detected a start bit, it need maintain synchronization only for the duration of a single character. The receiver examines successive received bits by sampling the incoming signal at the center of each pulse. Because the clock at the receiver is not synchronized with the clock at the transmitter, each received data bit will not be sampled exactly at its center.

Figure 8.25 provides the internal arrangement of the ACIA, a highly programmable interface whose parameters can be defined under software control. The ACIA has a single receiver input pin and a single transmitter output pin. The ACIA's CPU side is very much the same as that of the PIA, as Figure 8.25 demonstrates. This ACIA doesn't have a RESET input (due to a lack of pins) and must be reset by a software command after the initial power-up sequence.

Figure 8.25 Organization of the ACIA

The ACIA's Peripheral Side Pins

The ACIA communicates with a peripheral via seven pins, which may be divided into three groups: receiver, transmitter, and modem control. At this point, all we need say is that the modem is a black box that interfaces a digital system to the public switched telephone network and therefore permits digital signals to be transmitted across the telephone system. A modem converts digital signals into audio (analog) tones. We'll look at the modem in more detail in chapter 11.

Receiver The receiver part of the ACIA has a clock input and a serial data input. The receiver clock is used to sample the incoming

data bits and may be 64, 16, or 1 times that of the bit rate of the received data; for example an ACIA operating at 9,600 bits/s might use a 16x receiver clock of 153,600 Hz. The serial data input receives data from the peripheral to which the ACIA is connected. Most systems require a special interface chip between the ACIA and the serial data link to convert the signal levels at the ACIA to the signal levels found on the data link.

Transmitter The transmitter part of the ACIA has a clock input from which it generates the timing of the transmitted data pulses. As in the case of the receiver, the transmitter clock may be operated at 64, 16, or 1 times the rate of the data. In many cases the transmitter and receiver clocks are derived from the same oscillator. The transmitter data output provides a serial signal at a TTL level.

Modem control The ACIA communicates with a *modem* or similar equipment via three active-low pins (two inputs and one output). The ACIA's *request to send* (RTS*) output may be set or cleared under software control and is used by the ACIA to tell the modem that it is ready to transmit data to it.

The two active-low inputs to the ACIA are *clear-to-send* (CTS*) and *data-carrier-detect* (DCD*). The CTS* input is a signal from the modem to the ACIA that inhibits the ACIA from transmitting data if the modem is not ready (because the telephone connection has not been established or has been broken). If the CTS* input is high, a bit is set in the ACIA's status register, indicating that the modem (or other terminal equipment) is not ready for data.

The modem uses the ACIA's DCD* input to tell the ACIA that the carrier has been lost (i.e., a signal is no longer being received) and that valid data is no longer available at the receiver's input. A low-to-high transition at the DCD* input sets a bit in the status register and may also initiate an interrupt if the ACIA is so programmed. In applications of the ACIA that don't use a modem, the CTS* and DCD* inputs are connected to a low-level and not used.

The ACIA's Internal Registers

The ACIA has four internal registers: a transmitter data register (TDR), a receiver data register (RDR), a control register (CR), and a status register (SR). Since the ACIA has a single register-select input RS, only two internal registers can be directly accessed by the CPU. However, as the status and receiver data registers are always read from, and the transmitter data register and control register are always written to, the ACIA's R/W* input is used to distinguish between the two pairs of registers. The addressing arrangement of the ACIA are given in Table 8.10.

Table 8.10 Register selection scheme of the ACIA

RS	R/W	Type of register	ACIA register
0	0	Write only	Control
0	1	Read only	Status
1	0	Write only	Transmitter data
1	1	Read only	Receiver data

The control register is a write-only register that defines the operational properties of the ACIA, particularly the format of the transmitted or received data. Table 8.11 defines the control register's format. The counter division field, CR_0 and CR_1, determines the relationship between the transmitter and receiver bit rates and their respective clocks (Table 8.12).

Table 8.11 Format of the ACIA's control register

7	6	5	4	3	2	1	0
Receive interrupt enable	Transmitter control		Word select			Counter division	

Table 8.12 Relationship between CR_1, CR_0 and the division ratio

CR_1	CR_0	Division ratio
0	0	¸ 1
0	1	¸ 16
1	0	¸ 64
1	1	Master reset

When CR_1 and CR_0 are both set to one, the ACIA is reset and all internal status bits, with the exception of the CTS and DCD flags, are cleared. The CTS and DCD flags are entirely dependent on the signal level at the respective pins. The ACIA is initialized by first

writing ones into bits CR_1 and CR_0 of the control register, and then writing one of the three division ratio codes into these positions. In the majority of systems $CR_1 = 0$ and $CR_0 = 1$ for a divide by 16 ratio.

The word select field, CR_2, CR_3, CR_4 defines the format of the received or transmitted characters. These three bits allow the selection of eight possible arrangements of number of bits per character, type of parity, and number of stop bits (Table 8.13). For example, if you require a word with 8 bits, no parity and 1 stop bit control bits CR_4, CR_3, CR_2 must set to 1,0,1.

Table 8.13 The word select bits

CR_4	CR_3	CR_2	Word length	Parity	Stop bits	Total bits
0	0	0	7	Even	2	11
0	0	1	7	Odd	2	11
0	1	0	7	Even	1	10
0	1	1	7	Odd	1	10
1	0	0	8	None	2	11
1	0	1	8	None	1	10
1	1	0	8	Even	1	11
1	1	1	8	Odd	1	11

The transmitter-control field, CR_5 and CR_6, determines the level of the request to send (RTS*) output, and the generation of an interrupt by the transmitter portion of the ACIA. Table 8.14 gives the relationship between these controls bits and their functions. RTS* can be employed to tell the modem that the ACIA has data to transmit.

Table 8.14 Function of transmitter control bits CR_5, CR_6

CR_6	CR_5	RTS	Transmitter interrupt
0	0	Low	Disabled
0	1	Low	Enabled
1	0	High	Disabled
1	1	Low	Disabled—a break level is placed on the transmitter output

The transmitter interrupt mechanism can be enabled or disabled depending on whether the CPU is operating in an interrupt-driven or in a polled data mode. If the transmitter interrupt is enabled, a transmitter interrupt is generated whenever the transmitter data register (TDR) is empty, signifying the need for new data from the CPU. If the ACIA's clear-to-send input is high, the TDR empty flag bit in the status register is held low, inhibiting any transmitter interrupt.

The effect of setting both CR_6 and CR_5 to a logical one requires some explanation. If both these bits are high a *break* is transmitted until the bits are altered under software control. That is, the transmitter output of the ACIA is held at its space level. A break can be used to generate an interrupt at the receiver because the asynchronous format of the serial data precludes the existence of a space level for more than about ten bit periods.

The receiver interrupt enable field consists of bit CR_7 which, when clear, inhibits the generation of interrupts by the receiver portion of the ACIA. Whenever bit CR_7 is set, a receiver interrupt is generated by the receiver data register (RDR) flag of the status byte going high, indicating the presence of a new character ready for the CPU to read. A receiver interrupt can also be generated by a low-to-high transition at the data-carrier-detect (DCD) input, signifying the loss of a carrier. CR_7 is a composite interrupt enable bit. It is impossible to enable either an interrupt caused by the RDR being empty or an interrupt caused by a positive transition on the DCD* pin alone.

Configuring the ACIA

The following 68000 assembly language listing demonstrates how the ACIA is initialized before it can be used to transmit and receive serial data.

```
* Setting up an ACIA
*
ACIA      EQU     $800000          Location of ACIA in memory
CR        EQU     0                Control register offset
          LEA     ACIA,A0          A0 points to ACIA
*
* Perform a software reset by writing 1,1 to CR1, CR0
```

```
          MOVE.B #%00000011,CR(A0)
*
* Select counter division ratio as clk/16 CR1,CR0 = 0,1
* Select character format CR4,CR3,CR2 = 1,0,1
* Select operating mode
*      CR6,CR5 = 0,1 = assert RTS and enable transmitter interrupt
* Select receiver interrupt mode CR7 = 1 to enable Rx interrupt
          MOVE.B #%10110101,CR(A0) Set up ACIA
```

The Status Register

The status register has the same address as the control register, but is distinguished from it by being a read-only register. Table 8.15 gives the format of the status register. Let's look at the function of these bits.

Table 8.15 Format of the ACIA's control register

7	6	5	4	3	2	1	0
IRQ	PE	OVRN	FE	CTS	DCD	TDRE	RDRF

Bit 0—Receiver Data Register Full (RDRF) When set the RDRF bit indicates that the receiver data register is full and a character has been received. If the receiver interrupt is enabled, the interrupt request flag, bit 7, is also set whenever RDRF is set. Reading the data in the receiver data register clears the RDRF bit. Whenever the DCD input is high, the RDRF bit remains at a logical zero, indicating the absence of any valid input.

The RDRF bit is used to detect the arrival of a character when the ACIA is operated in a polled input mode.

```
*  Subroutine to receive a character
*   REPEAT
*     Read ACIA status
*   UNTIL RDRF = 1
*  Read ACIA data
*
ACIA      EQU     $800000
RDRF      EQU     0             Rx data ready = bit 0 of SR
SR        EQU     2             Offset for status register
DR        EQU     0             Offset for data register
          LEA     ACIA,A0       A0 points to ACIA
POLL      TST.B   #RDRF,SR(A0)  REPEAT Test Rx status bit
          BEQ     POLL          UNTIL character received
          MOVE.B  DR(A0),D0     Move input from ACIA to D0
          RTS
```

Bit—1 Transmitter Data Register Empty (TDRE) This flag is the transmitter counterpart of RDRF. A logical 1 in TDRE indicates that the contents of the transmitter data register (TDR) have been transmitted and the register is now ready for new data. The IRQ bit is also set whenever the TDRE flag is set if the transmitter interrupt is enabled. The TDRE bit is at a logical zero when the TDR is full, or when the CTS input is high, indicating that the terminal equipment is not ready for data. The fragment of code below demonstrated how the TDRE flag is used when the ACIA is operated in a polled output mode.

```
* Subroutine to transmit a character
* REPEAT
*   Read ACIA status
* UNTIL TDRE = 1
* Write data to ACIA
*
ACIA      EQU    $800000
TDRE      EQU    1             Transmitter data register empty = bit 1
SR        EQU    2             Offset for status register
DR        EQU    0             Offset for data register
          LEA    ACIA,A0       A0 points to ACIA base
POLL      BTST.B #TDRE,SR(A0)  Test transmitter for empty state
          BEQ    POLL          Repeat until transmitter ready
          MOVE.B D0,DR(A0)     Move byte from D0 to ACIA
          RTS
```

Bit—2 Data Carrier Detect (DCD) The *data-carrier-detect* bit is set whenever the DCD* input is high, indicating that a carrier is not present. The DCD* pin is normally employed only in conjunction with a modem. When the signal at the DCD* input makes a low-to-high transition, the DCD bit in the status register is set and the IRQ bit is also set if the receiver interrupt is enabled. The DCD bit remains set even if the DCD* input returns to a low state. To clear the DCD bit, the CPU must read the contents of the ACIA's status

register and then the contents of the data register.

Bit—3 Clear to Send (CTS) The *clear-to-send* bit directly reflects the status of the ACIA's CTS* input. A low level on the CTS* input indicates that the modem is ready for data. If the CTS bit is set, the transmitter data register empty bit is inhibited (clamped at zero), and no data may be transmitted by the ACIA.

Bit—4 Framing Error (FE) The *framing error* bit is set whenever a received character is incorrectly framed by a start bit and a stop bit. A framing error is detected by the absence of the first stop bit and indicates a synchronization (timing) error, a faulty transmission, or a break condition. The framing error flag is set or cleared during receiver data transfer time and is present throughout the time that the associated character is available.

Bit—5 Receiver Overrun (OVRN) The receiver overrun flag bit is set when a character is received, but hasn't been read by the CPU before a subsequent character is received. The new character over-writes the previous character which is now lost. Consequently, the receiver overrun bit indicates that one or more characters in the data stream have been lost. Synchronization is not affected by an overrun error—the error is caused by the CPU not reading a character, rather than by a fault in the transmission process. The overrun bit is cleared after reading the data from the RDR or by a master reset. Modern ACIAs usually have FIFO buffers to hold several characters to give the CPU more time to read them.

Bit—6 Parity Error (PE) The parity error is set whenever the received parity bit does not agree with the parity bit generated locally at the receiver from the preceding data bits. Odd or even parity may be selected by writing the appropriate code into bits 2, 3, and 4 of the control register. If no parity is selected, then both the transmitter parity generator and the receiver parity checker are disabled. Once a parity error has been detected and the parity error bit set, it remains set as long as a character with a parity error is in the receiver data register.

Bit—7 Interrupt Request (IRQ) The *interrupt request* bit is a composite interrupt request flag because it is set whenever the ACIA wishes to interrupt the CPU, for whatever reason. The IRQ bit may be set by any of the following:

- Receiver data register full (SR bit 0 set);
- Transmitter data register empty (SR bit 1 set);
- DCD bit set (SR bit 2).

Whenever IRQ = 1 the ACIA's IRQ* pin is forced active-low to request an interrupt from the CPU. The IRQ bit is cleared by a read from the receiver data register or a write to the transmitter data.

Programming the ACIA

We are now going to look at a more complete program that uses some of the ACIA's error detecting facilities when receiving data.

```
ACIAC   EQU     $800000     Base address of ACIA
ACIAD   EQU     ACIA+2      Address of data register
RDRF    EQU     0           Receiver data register full
TDRE    EQU     1           Transmitter data register empty
DCD     EQU     2           Data carrier detect
CTS     EQU     3           Clear to send
FE      EQU     4           Framing error
OVRN    EQU     5           Over run
PE      EQU     6           Parity error

INPUT   MOVE.B  ACIAC,D0    Get status from ACIA
        BTST    #RDRF,D0    Test for received character
        BNE     ERROR_CK    If character received then test SR
        BTST    #DCD,D0     Else test for loss of signal
        BEQ     INPUT       Repeat loop while DCD clear
        BRA     DCD_ERR     Else deal with loss of signal
ERR_CK  BTST    #FE,D0      Test for framing error
        BNE     FE_ERR      If framing error, deal with it
        BTST    #OVRN,D0    Test for overrun
        BNE     OV_ERR      If overrun, deal with it
        BTST    #PE,D0      Test for parity error
        BNE     PE_ERR      If parity error deal with it
        MOVE.B  ACIAD,D0    Load the input into D0
        BRA     EXIT
*
DCD_ERR                     Deal with loss of signal
        BRA     EXIT
*
FE_ERR  Deal with framing error
```

```
        BRA     EXIT
*
OV_ERR  Deal with overrun error
        BRA     EXIT
*
PE_ERR  Deal with parity error
*
EXIT    RTS
```

So far we've examined how information in digital form is read by a computer, processed in the way dictated by a program and then output in digital form. We haven't yet considered how information is converted between real-world form and digital form. In the next section we describe some of the most frequently used computer interfaces such as the keyboard, the display and the printer.