- Supervisory control.
- Human–computer interfacing.

These activities are to be found in one form or another in all embedded computer applications and so although in this chapter we have concentrated on process control applications the ideas (and problems) are common to a wide range of other applications.

Also covered were the ways in which several computers can be configured for control applications. These include dual computer systems to increase reliability, and distributed and hierarchical configurations. A brief mention was made of some advanced control strategies.

EXERCISES

2.1 List the characteristics of (a) batch processes and (b) continuous processes.

2.2 You are the manager of a plant which can produce ten different chemical products in batches which can be between 500 and 5000 kg. What factors would you expect to consider in calculating the optimum batch size? What arguments would you put forward to justify the use of an on-line computer to calculate optimum batch size?

2.3 What are the advantages/disadvantages of using a continuous oven? How will the control of the process change from using a standard oven on a batch basis to using an oven in which the batch passes through on a conveyor belt? Which will be the easier to control?

2.4 List the advantages and disadvantages of using DDC.

2.5 List the advantages of using several small computers instead of one large computer in control applications. Are there any disadvantages that arise from using several computers?

2.6 In the section on human–computer interfacing we made the statement 'the design of user interfaces is a specialist area'. Can you think of reasons to support this statement and suggest what sort of background and training a specialist in user interfaces might require?

3

Computer Hardware Requirements for Real-time Applications

This chapter provides a brief overview of some of the basic ideas relating to computer hardware. A brief description of the various types of computers such as microprocessors, microcomputers and special purpose computers is given. A detailed explanation of the standard methods for data transfer including consideration of the use of interrupts is provided. Also given is a brief overview on communication methodologies. The emphasis throughout is on the principles involved and not on the characteristics of a particular microprocessor or microprocessor support chips.

The aims of the chapter are to provide:

- A basic description of the major features of microprocessors.
- A description of the standard interfacing techniques.
- An overview of the standard communication methodologies.

3.1 INTRODUCTION

Although almost any digital computer can be used for real-time computer control and other real-time operations, they are not all equally easily adapted for such work. In the majority of embedded computer-based systems the computer used will be a microprocessor, a microcomputer or a specialised digital processor. Specialised digital processors include fast digital signal processors, parallel computers such as the transputer, and special RISC (Reduced Instruction Set Computers) for use in safety-critical applications (for example, the VIPER (Cullyer and Pygott, 1987)).

3.2 GENERAL PURPOSE COMPUTER

The general purpose microprocessors include the Intel XX86 series, Motorola 680XX series, National 32XXX series and the Zilog Z80 and Z8000 series.

A characteristic of computers used in control systems is that they are modular: they provide the means of adding extra units, in particular specialised input and output devices, to a basic unit. The capabilities of the basic unit in terms of its processing power, storage capacity, input/output bandwidth and interrupt structure determine the overall performance of the system. A simplified block diagram of the basic unit is shown in Figure 3.1; the arithmetic and logic, control, register, memory and input/output units represent a general purpose digital computer.

Of equal importance in a control computer are the input/output channels which provide a means of connecting process instrumentation to the computer, and also the displays and input devices provided for the operator. The instruments are not usually connected directly but by means of interface units. Also of importance is the

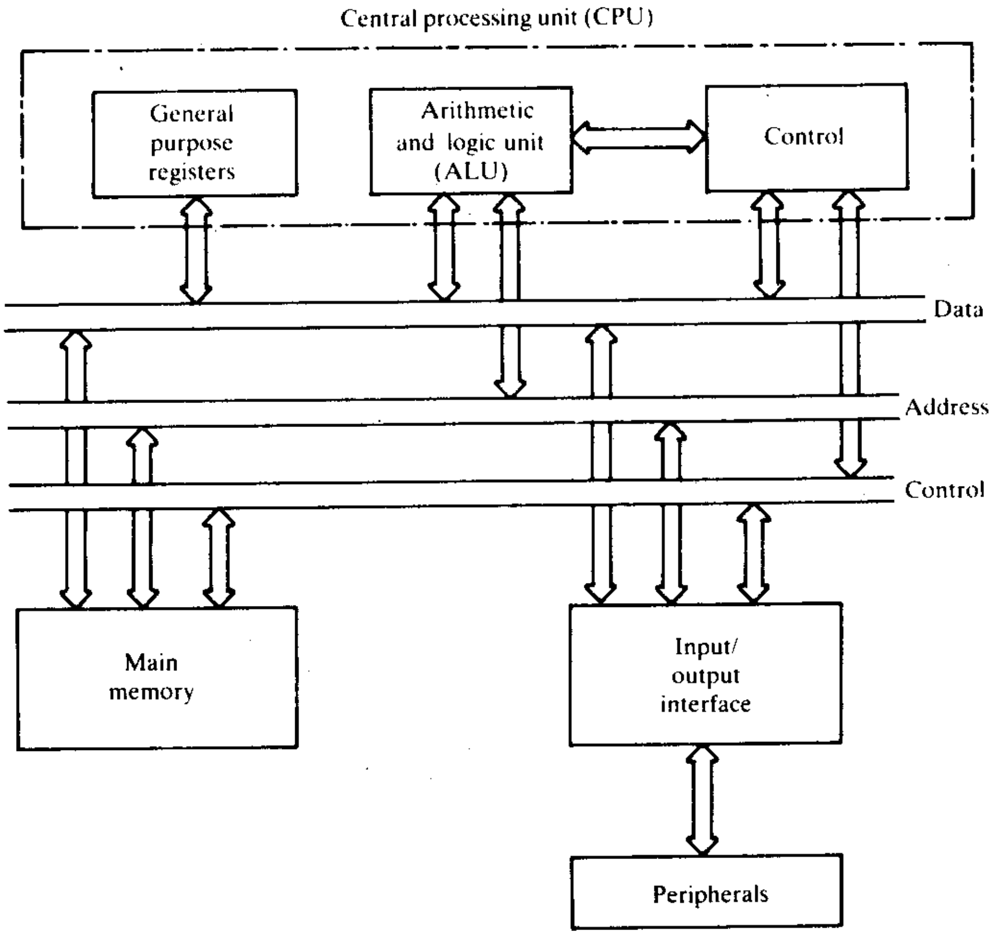
Central processing unit (CPU)
General purpose registers
Arithmetic and logic unit (ALU)
Control
Data
Address
Control
Main memory
Input/ output interface
Peripherals

Figure 3.1 Schematic diagram of a general purpose digital computer.

ability to communicate with other computers since, as we discussed in the previous chapter, many modern computer control systems involve the use of several interconnected computers.

3.2.1 Central Processing Unit

The arithmetic and logic unit (ALU) together with the control unit and the general purpose registers make up the central processing unit (CPU). The ALU contains the circuits necessary to carry out arithmetic and logic operations, for example to add numbers, subtract numbers and compare two numbers. Associated with it may be hardware units to provide multiplication and division of fixed point numbers and, in the more powerful computers, a floating point arithmetic unit. The general purpose registers can be used for storing data temporarily while it is being processed. Early computers had a very limited number of general purpose registers and hence frequent access to main memory was required. Most computers now have CPUs with several general purpose registers – some large systems have as many as 256 registers – and for many computations, intermediate results can be held in the CPU without the need to access main memory thus giving faster processing.

The control unit continually supervises the operations within the CPU: it fetches program instructions from main memory, decodes the instructions and sets up the necessary data paths and timing cycles for the execution of the instructions.

The features of the CPU which determine the processing power available and hence influence the choice of computer for process control include:

- wordlength;
- instruction set;
- addressing methods;
- number of registers;
- information transfer rates; and
- interrupt structure.

The computer wordlength is important both in ensuring adequate precision in calculations and in allowing direct access to a large area of main storage within one instruction word. It is possible to compensate for short wordlengths, both for arithmetic precision and for memory access, by using multiple word operations, but the penalty is increased time for the operations.

The basic instruction set of the CPU is also important in determining its overall performance. Features which are desirable are:

- flexible addressing modes for direct and immediate addressing;
- relative addressing modes;
- address modification by use of index registers;
- instructions to transfer variable length blocks of data between storage units or locations within memory; and
- single commands to carry out multiple operations.

These features reduce the number of instructions required to perform 'housekeeping' operations and hence both reduce storage requirements and improve overall speed of operation by reducing the number of accesses to main memory required to carry out the operations. A consequence of an extensive and powerful instruction set is, however, that efficient programming in the assembly language becomes more difficult because the language can become complex; thus it is desirable to be able to program the system using a high-level language which has a compiler designed to make optimum use of the special features of the instruction set. There are many other reasons for *not* using assembly languages and we will discuss some of them in Chapter 5.

Another area which must be considered carefully when selecting a computer for process control is information transfer, both within the CPU and between the backing store and the CPU, and also with the input/output devices. The rate at which such transfers can take place, the ability to carry out operations in parallel with the processing of data, and the ability to communicate with a large range of devices can be crucial to the application to process control. A vital requirement is also a flexible and efficient multi-level interrupt structure.

3.2.2 Storage

The storage used on computer control systems divides into two main categories: fast access storage and auxiliary storage. The fast access memory is that part of the system which contains data, programs and results which are currently being operated on. The major restriction with current computers is commonly the addressing limit of the processor. In addition to RAM (random access memory – read/write) it is now common to have ROM (read-only memory), PROM (programmable read-only memory) or EPROM (electronically programmable read-only memory) for the storage of critical code or predefined functions.

The use of ROM has eased the problem of memory protection to prevent loss of programs through power failure or corruption by the malfunctioning of the software (this can be a particular problem during testing). An alternative to using ROM is the use of memory mapping techniques that trap instructions which attempt to store in a protected area. This technique is usually only used on the larger systems which use a memory management system to map program addresses onto the physical address space. An extension of the system allows particular parts of the physical memory to be set as read only, or even locked out altogether: write access can be gained only by the use of 'privileged' instructions.

The auxiliary storage medium is typically disk or magnetic tape. These devices provide bulk storage for programs or data which are required infrequently at a much lower cost than fast access memory. The penalty is a much longer access time and the need for interface boards and software to connect them to the CPU. Auxiliary or backing store devices operate asynchronously to the CPU and care has to be taken in deciding on the appropriate transfer technique for data between the CPU, fast access memory and the backing store. In a real-time system use of the CPU to carry out the transfer is not desirable as it is slow and no other computation can take place during transfer. For efficiency of transfer it is sensible to transfer large blocks of data rather than a single word or byte and this can result in the CPU not being available for up to several seconds in some cases.

The approach frequently used is direct memory access (DMA). For this the interface controller for the backing memory must be able to take control of the address and data buses of the computer.

3.2.3 Input and Output

The input/output (I/O) interface is one of the most complex areas of a computer system; part of the complication arises because of the wide variety of devices which have to be connected and the wide variation in the rates of data transfer. A printer may operate at 300 baud whereas a disk may require a rate of 500 kbaud. The devices may require parallel or serial data transfers, analog-to-digital or digital-to-analog conversion, or conversion to pulse rates.

The I/O system of most control computers can be divided into three sections:

- process I/O;
- operator I/O; and
- computer I/O.

3.2.4 Bus Structure

Buses are characterised in three ways:

- mechanical (physical) structure;
- electrical; and
- functional.

In mechanical or physical terms a bus is a collection of conductors which carry electrical signals, for example tracks on a printed circuit board or the wires in a ribbon cable. The physical form of the bus represents the *mechanical characteristic* of the bus system. The *electrical characteristics* of the bus are the signal levels, loading (that is, how many loads the line can support), and type of output gates (open-collector, tri-state). The *functional characteristics* describe the type of information which the electrical signals flowing along the bus conductors represent. The bus lines can be divided into three functional groups:

- address lines;
- data lines; and
- control and status lines.

These can be thought of as ***where***, ***what*** and ***when***. The address lines provide information on where the information is to be sent (or where it is to be obtained from); the data lines show what the information is; and the control and status lines indicate when it is to be sent.

3.3 SINGLE-CHIP MICROCOMPUTERS AND MICROCONTROLLERS

Many integrated circuit manufacturers produce microcomputers in which all the components necessary for a complete computer are provided on one single chip. A typical single-chip device is shown in Figure 3.2. With only a small amount of EPROM and an even smaller amount of RAM this type of device is obviously intended for small, simple systems. The memory can always be extended by using external memory chips.

The microcontroller is similarly a single-chip device that is specifically intended for embedded computer control applications. The main difference between it and a microcomputer is that it typically will have on board the chip a multiplexed ADC and some form of process output, for example a pulse width modulator unit. The chip may also contain a real-time clock generator and a watch-dog timer.

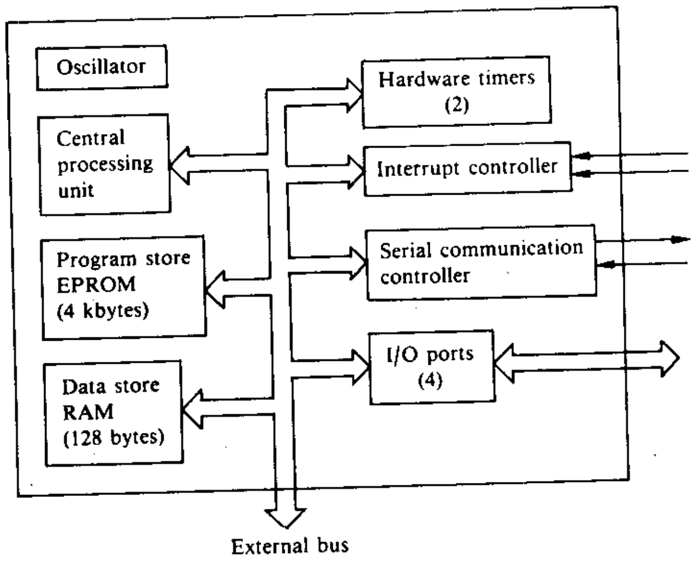

Figure 3.2 A typical single-chip computer.

3.4 SPECIALISED PROCESSORS

Specialised processors have been developed for two main purposes:

- safety-critical applications; and
- increased computation speed.

For safety-critical applications the approach has been to simplify the instruction set – the so-called reduced instruction set computer (RISC). The advantage of simplifying the instruction set is the possibility of formal verification (using mathematical proofs) that the logic of the processor is correct. The second advantage of the RISC machine is that it is easier to write assemblers and compilers for the simple instruction set. An example of such a machine is the VIPER (Cullyer, 1988; Dettmer, 1986), the main features of which are:

- Formal mathematical description of the processor logic.
- Integer arithmetic (32 bit) and no floating point operations (it is argued that floating point operations are inexact and cannot be formally verified).
- No interrupts – all event handling is done using polling (again interrupts make formal verification impossible).
- No dynamic memory allocation.

(There is an unresolved dispute regarding the validity and completeness of the formal verification procedures used for the VIPER processor (MacKenzie, 1993).)

The traditional Von Neumann computer architecture with its one CPU through which all the data and instructions have to pass sequentially results in a bottleneck. Increasing the processor speed can increase the throughput but eventually systems will reach a physical limit because of the fundamental limitation on the speed at which an electronic signal can travel. The search for increased processing speed has led to the abandonment of the Von Neumann architecture for high-speed computing.

3.4.1 Parallel Computers

Many different forms of parallel computer architectures have been devised; however, they can be summarised as belonging to one of three categories:

SIMD	Single instruction stream, multiple data stream.
MISD	Multiple instruction stream, single data stream.
MIMD	Multiple instruction stream, multiple data stream.

These are illustrated in Figure 3.3 where the traditional architecture characterised as SISD (Single instruction stream, single data stream) is also shown.

MIMD systems are obviously the most powerful class of parallel computers in that each processor can potentially be executing a different program on a different data set. The most widely available MIMD system is the INMOS transputer. Each

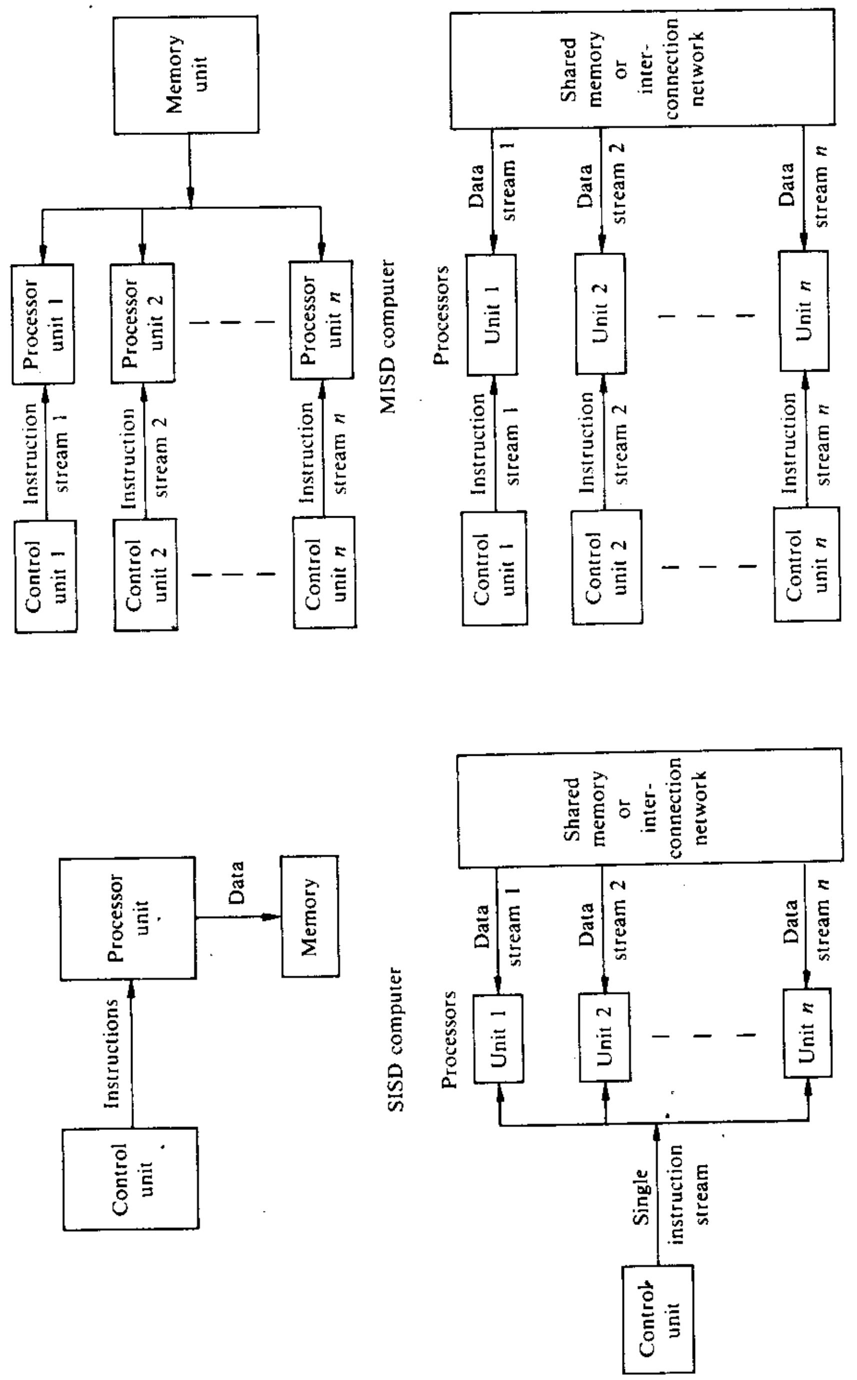

Figure 3.3 Computer system architectures.

transputer chip has a CPU, on-board memory, an external memory interface and communication links for direct point-to-point connection to other transputer chips. An individual chip can be used as a stand-alone computing device; however, the power of the transputer is obtained when several transputers are interconnected to form a parallel processing network.

INMOS developed a special programming language, occam, for use with the transputer. Occam is based on the assumption that the application to be implemented on the transputer can be modelled as a set of processes (actions) that communicate with each other via channels. A channel is a unidirectional link between two processes which provides synchronised communication. A process can be a primitive process, or a collection of processes; hence the system supports a hierarchical structure. Processes are dynamic in that they can be created, can die and can create other processes.

3.4.2 Digital Signal Processors

In applications such as speech processing, telecommunications, radar and hi-fi systems analog techniques have been used for modifying the signal characteristics. There are advantages to be gained if such processing can be done using digital techniques in that the digital devices are inherently more reliable and not subject to drift. The problem is that the bandwidth of the signals to be processed is such as to demand very high processing speeds.

Special purpose integrated circuits optimised to meet the signal processing requirements have been developed. They typically use the so-called Harvard architecture in which separate paths are provided for data and for instructions. DSPs typically use fixed point arithmetic and the instruction set contains instructions for manipulating complex numbers. They are difficult to program as few high-level language compilers are available.

3.5 PROCESS-RELATED INTERFACES

Instruments and actuators connected to the process or plant can take a wide variety of forms: they may be used for measuring temperatures and hence use thermocouples, resistance thermometers, thermistors, etc.; they could be measuring flow rates and use impulse turbines; they could be used to open valves or to control thyristor-operated heaters. In all these operations there is a need to convert a digital quantity, in the form of a bit pattern in a computer word, to a physical quantity, or to convert a physical quantity to a bit pattern. Designing a different interface for each specific type of instrument or actuator is not sensible or economic and hence we look for some commonality between them. Most devices can be allocated to one of the following four categories:

1. *Digital quantities*: These can be either binary, that is a valve is open or closed, a switch is on or off, a relay should be opened or closed, or a generalised digital quantity, that is the output from a digital voltmeter in BCD (binary coded decimal) or other format.
2. *Analog quantities*: Thermocouples, strain gauges, etc., give outputs which are measured in millivolts; these can be amplified using operational amplifiers to give voltages in the range -10 to $+10$ volts; conventional industrial instruments frequently have a current output in the range 4 to 20 mA (current transmission gives much better immunity to noise than transmission of low-voltage signals). The characteristic of these signals is that they are continuous variables and have to be both sampled and converted to a digital value.
3. *Pulses and pulse rates*: A number of measuring instruments, particularly flow meters, provide output in the form of pulse trains; similarly the increasing use of stepping motors as actuators requires the provision of pulse outputs. Many traditional controllers have also used pulse outputs: for example, valves controlling flows are frequently operated by switching a dc or ac motor on and off, the length of the on pulse being a measure of the change in valve opening required.
4. *Telemetry*: The increasing use of remote outstations, for example electricity substations and gas pressure reduction stations, has increased the use of telemetry. The data may be transmitted by landline, radio or the public telephone network: it is, however, characterised by being sent in serial form, usually encoded in standard ASCII characters. For small quantities of data the transmission is usually asynchronous. Telemetry channels may also be used on a plant with a hierarchy of computer systems instead of connecting the computers by some form of network. An example of this is the CUTLASS system used by the Central Electricity Generating Board, which uses standard RS232 lines to connect a hierarchy of control computers.

The ability to classify the interface requirements into the above categories means that a limited number of interfaces can be provided for a process control computer. The normal arrangement is to provide a variety of interface cards which can be added to the system to make up the appropriate configuration for the process to be controlled; for example, for a process with a large number of temperature measurements several analog input boards may be required.

3.5.1 Digital Signal Interfaces

A simple digital input interface is shown in Figure 3.4. It is assumed that the plant outputs are logic signals which appear on lines connected to the digital input

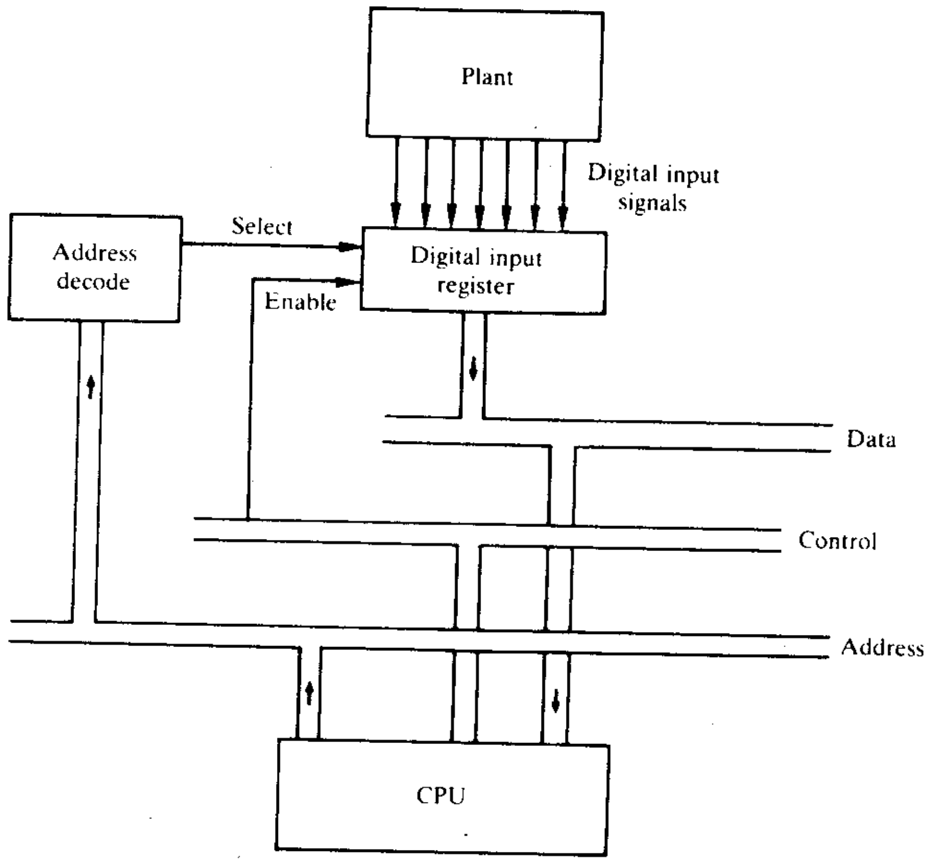

Figure 3.4 Simple digital input interface.

register. It is usual to transfer one word at a time to the computer, so normally the digital input register will have the same number of input lines as the number of bits in the computer word. The logic levels on the input lines will typically be 0 and + 5 V; if the contacts on the plant which provide the logic signals use different levels then conversion of signal levels will be required.

To read the lines connected to the digital input register the computer has to place the address of the register on the address bus and decoding circuitry is required in the interface (address decoder) to select the digital input register. In addition to the 'select' signal an 'enable' signal may also be required; this could be provided by the 'read' signal from the computer control bus. In response to both the 'select' and 'enable' signals the digital input register enables its output gates and puts data onto the computer data bus. Note that for proper operation of the data bus the digital input register must connect its output gates to the data bus only when it is selected and enabled; if it connects at any other time it will corrupt data intended for other devices.

The timing of the transfer of information will be governed by the CPU timing. A typical example is shown in Figure 3.5. For this system it is assumed that the transfer requires three cycles of the system clock, labelled T_1, T_2, and T_3. The address lines begin to change at the beginning of the cycle T_1 and they are guaranteed to be valid by the start of cycle T_2; also at the start of cycle T_2 the READ line becomes active. For the correct read operation the digital input register has to provide stable data at the negative-going edge (or earlier) of the clock during the T_3 cycle and the data must remain on the data lines until the negative-going edge of the following clock cycle. Note that the actual time taken to transfer the data from the data bus to the CPU may be much shorter than the time for which the data is valid. The requirement that it remain valid from the negative-going edge of cycle T_3 until the negative-going edge of the following cycle is to provide for the worst case condition arising from variations in the performance of the various components.

Figure 3.4 shows a system that provides information only on demand from the computer: it cannot indicate to the computer that information is waiting. There are many circumstances in which it is useful to indicate a change of status of input lines to the computer. To do this a status line which the computer can test, or which can be used as an interrupt, is needed.

A simple digital output interface is shown in Figure 3.6. Digital output is the simplest form of output: all that is required is a register or latch which can hold the

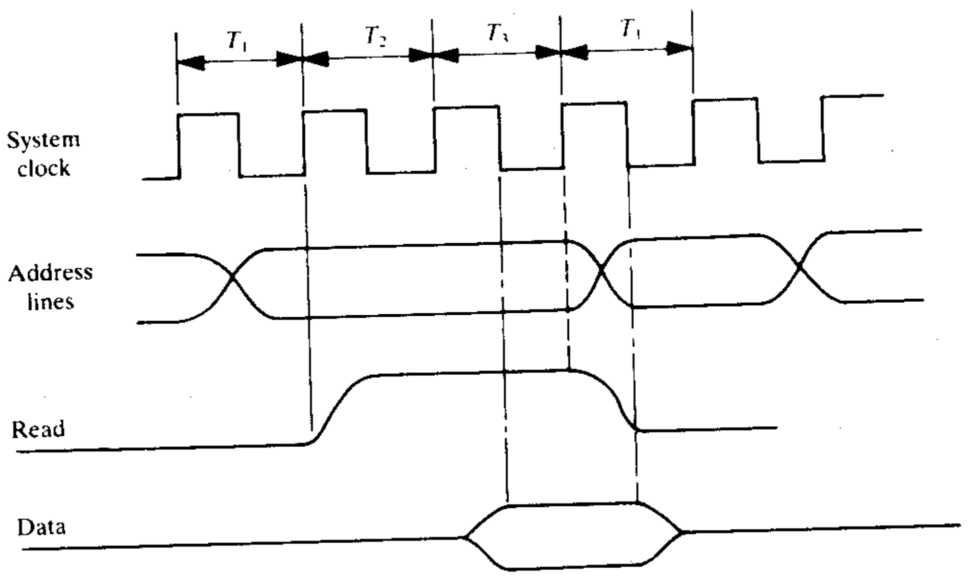

Figure 3.5 Simplified READ (INPUT) timing diagram.

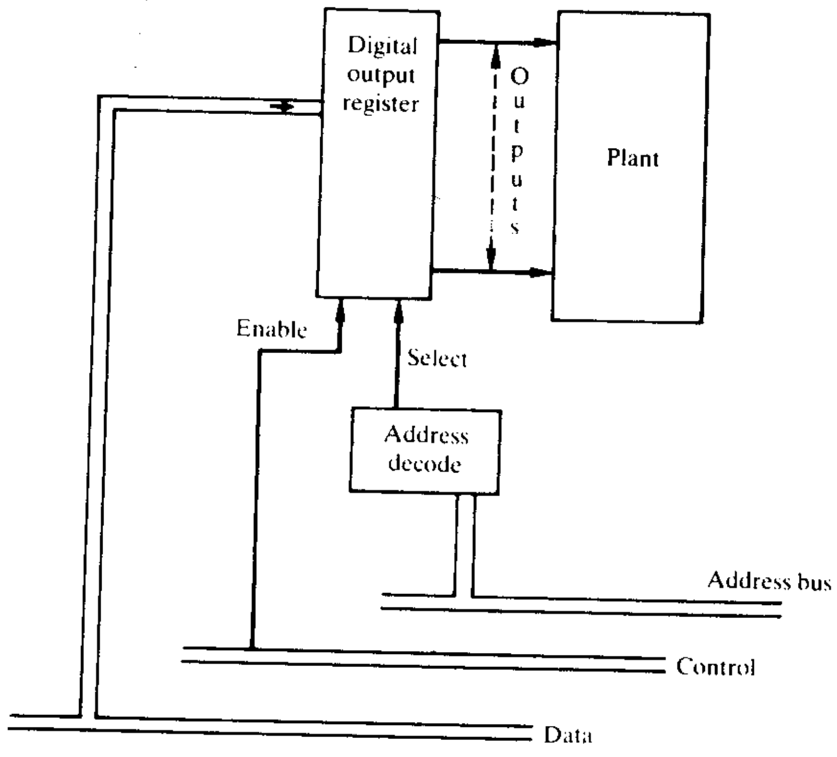

Figure 3.6 Simple digital output interface.

data output from the computer. To avoid the data in the register changing when the data on the data bus changes, the output latch must respond only when it is addressed. The 'enable' signal is used to indicate to the device that the data is stable on the data bus and can be read. The latch must be capable of accepting the data in a very short length of time, typically less than 1 microsecond.

The output from the latch is a set of logic levels, typically 0 to +5 V: if these levels are not adequate to operate the actuators on the plant, some signal conversion is necessary. This conversion is often performed by using the low-level signals to operate relays which carry the higher-voltage signals: an advantage which is gained from the use of relays is that there is electrical isolation between the plant and the computer system. The relay can be a mechanical device or more commonly now an optical isolation device.

The digital input and output interfaces described above can also be used to accept BCD data from instruments, since they are essentially parallel digital input and output devices. A 16 bit digital input device could, for example, transmit four BCD digits to the computer (this would correspond to a precision of one part in 10 000 – 0 to 9999).

Because digital input and output is a frequently required operation, many microprocessor manufacturers produce integrated circuits which provide such an interface.

3.5.2 Pulse Interfaces

In its simplest form a pulse input interface consists of a counter connected to a line from the plant. The counter is reset under program control and after a fixed length of time the contents are read by the computer. A typical arrangement is shown in Figure 3.7, which also shows a simple pulse output interface. The transfer of data from the counter to the computer uses techniques similar to those for the digital input described above.

The measurement of the length of time for which the count proceeds can be

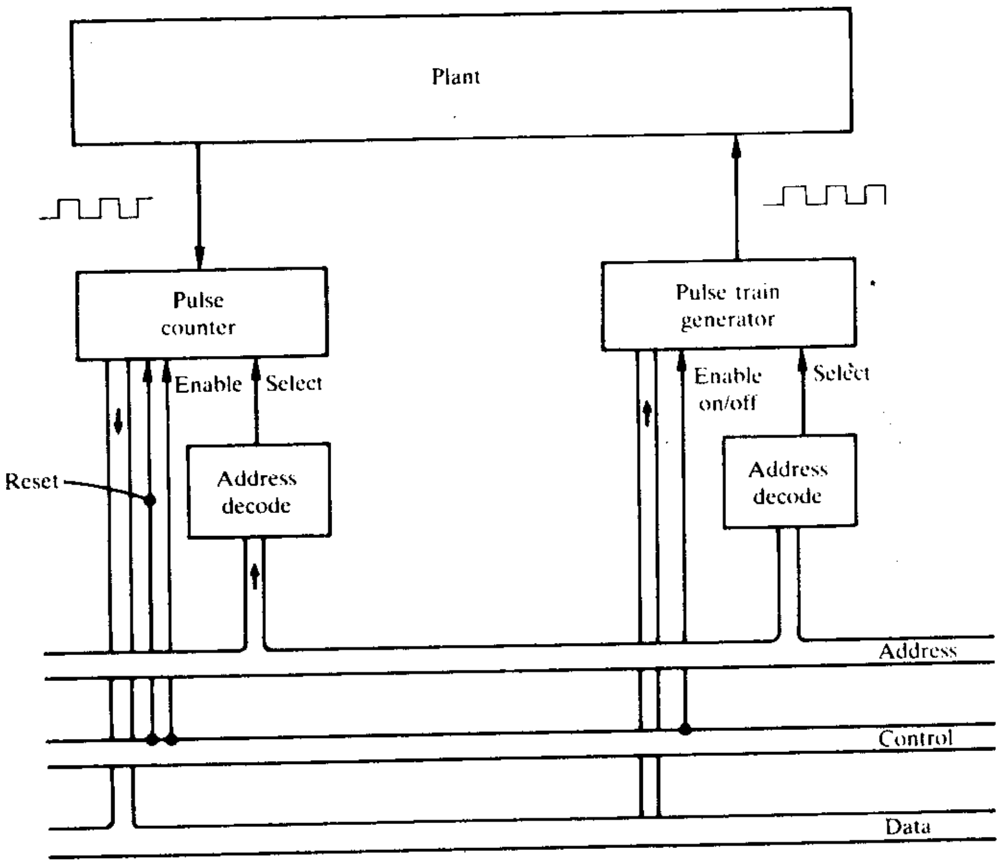

Figure 3.7 Pulse input and output interface.

carried out either by a logic circuit in the counter interface or by the computer. If the timing is done by the computer then the 'enable' signal must inhibit the further counting of pulses. If the computing system is not heavily loaded, the external interface hardware required can be reduced by connecting the pulse input to an interrupt and counting the pulses under program control.

Pulse outputs can take a variety of forms:

1. a series of pulses of fixed duration;
2. a single pulse of variable length (time-proportioned output); and
3. pulse width modulation – a series of pulses of different widths sent at a fixed frequency.

For type 1 the computer turns a pulse generator on or off, or loads a register with the number of pulses to be transmitted. The pulse output is sent to the process and used to decrement the register contents; when the register reaches zero the pulse output is turned off. A system of this type could be used, for example, to control the movement of a stepping motor.

For type 2 the computer raises or lowers a logic line and thus sends a variable length pulse to the plant, or loads a register with a number specifying the length of pulse required and interface logic is used to generate the pulse. The variable length pulse system is used typically to operate process control valves. Using the computer to turn the pulse train on or off directly is usual only on small systems with a few input or output lines, or when the pulse rate is low. For large systems, or for high pulse rates, it is normal to arrange for the interface logic to generate the actual pulse train or to control the duration of the pulse.

For type 3, pulse width modulation (PWM), special purpose interface chips are used to generate the pulses. Normally with this type of output a fast PWM stream will be produced and this is then converted into a linear analog output by using a low-pass filter.

Closely related to pulse counters are *hardware timers*. If the pulse counter is made to count down from a preset value at a fixed rate it can act as a timer. For example, if the clock rate used to decrement the counter is set at one count per millisecond, it can be used as a timer with a precision of 1 ms. The counter is loaded with a binary number corresponding to the desired time interval and the count started; when the counter reaches zero it generates an interrupt to indicate that it has 'timed out', that is that the interval has elapsed. The computer can then take the appropriate action.

The input to a hardware timer is normally a continuously running accurate pulse generator which either may have a fixed frequency or may be programmable to give a range of frequencies, for example thousandths, hundredths, tenths and seconds. The unit is programmed either by setting external switches or by commands sent by the computer. Hardware timers can be used to set the maximum time allowed for the response from an external device: the computer requests a response from a device and at the same time starts a hardware timer; if the device has not responded by the time the hardware timer interrupts then an error signal is generated.

A special form of this is the *watch-dog timer* which is often used on process control computers. The timer is reset at fixed intervals, usually when the operating system kernel is entered: if the watch-dog timer 'times out' it indicates that for some reason the operating system kernel has not been entered at the correct time, either because of some hardware malfunction, or because the normal interrupts have been locked out by a software error. A hardware timer can be used as a real-time clock (see section 3.5.4 below).

3.5.3 Analog Interfaces

The conversion of analog measurements to digital measurements involves two operations: sampling and quantisation. The sampling rate necessary for controlling a process is discussed in the next chapter. As is shown in Figure 3.8 many analog-to-digital converters (ADCs) include a 'sample–hold' circuit on the input to the device. The sample time of this unit is much shorter than the sample time required for the process; this sample–hold unit is used to prevent a change in the quantity being measured while it is being converted to a discrete quantity.

To operate the analog input interface the computer issues a 'start' or 'sample' signal, typically a short pulse (1 microsecond), and in response the ADC switches the 'sample–hold' into SAMPLE for a short period after which the quantisation process commences. Quantisation may take from a few microseconds to several milliseconds. On completion of the conversion the ADC raises a 'ready' or 'complete' line which is either polled by the computer or is used to generate an interrupt.

Use of separate ADCs for each analog input is expensive, despite the reduction in price in recent years, and typically a multiplexer is used to switch the inputs from several input lines to a single ADC (see Figure 3.8). For high-level (0–10 V) signals the multiplexer is usually a solid-state device (typically based on the use of field effect transistor switches); for low-level signals in the millivolt range, for example from thermocouples or strain gauges, mercury-wetted reed-relay switch units are used. For low-level signals, a programmable gain amplifier is usually used between the multiplexer and the sample–hold unit. With a multiplexed system the sequence of operations is more complex than with a single-channel device as the program has to arrange for the selection of the appropriate input channel.

For the simplest systems a single channel-select signal is used which causes the multiplexer to step to the next channel: the channels are thus sampled in sequence. A more elaborate arrangement is to provide random channel selection by connecting channel address inputs to the computer data bus. The sequence of events is then: select the channel address, send the start conversion command and then wait for the conversion complete signal. In some high-speed converters it is possible to send the next channel address during the period in which the present input is being quantised. This technique is also frequently used with reed-relay switching since a delay to allow time for the signal to stabilise is required between selecting a channel and sampling.

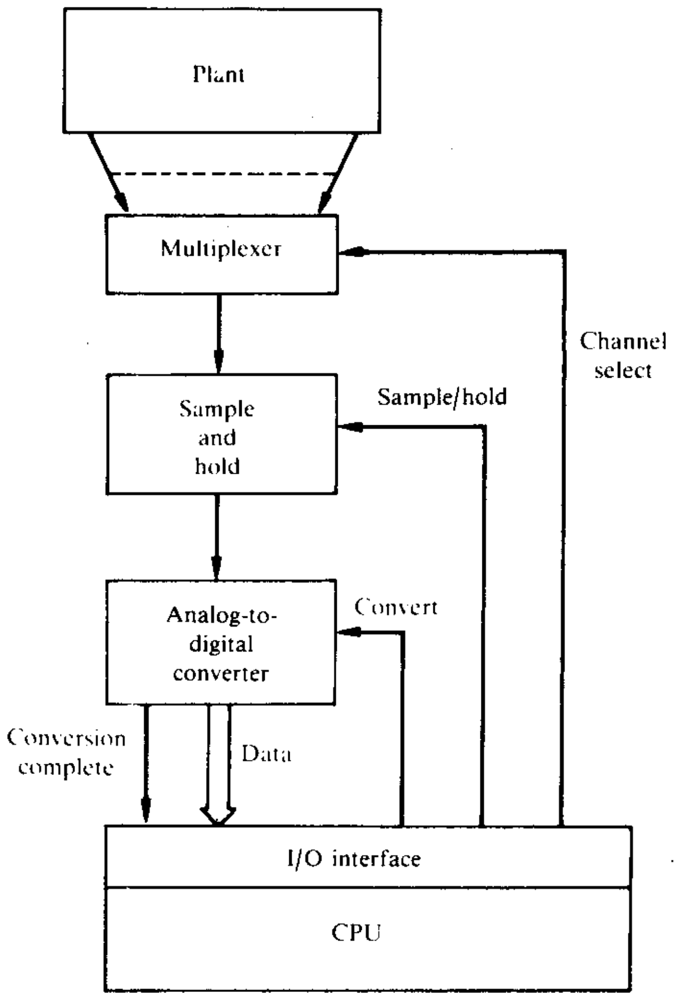

Figure 3.8 Analog input system.

Digital-to-analog conversion is simpler (and hence cheaper) than analog-to-digital conversion and as a consequence it is normal to provide one converter for each output. (It is possible to produce a multiplexer in order to use a single digital-to-analog converter (DAC) for analog output. Why would this solution not be particularly useful?) Figure 3.9 shows a typical arrangement. Each DAC is connected to the data bus and the appropriate channel is selected by putting the channel address on the computer address bus. The DAC acts as a latch and holds the previous value sent to it until the next value is sent. The conversion time is typically from 5 to 20 ms and typical analog outputs are −5 to +5 V, −10 to +10 V, or a current output of 0 to 20 mA.

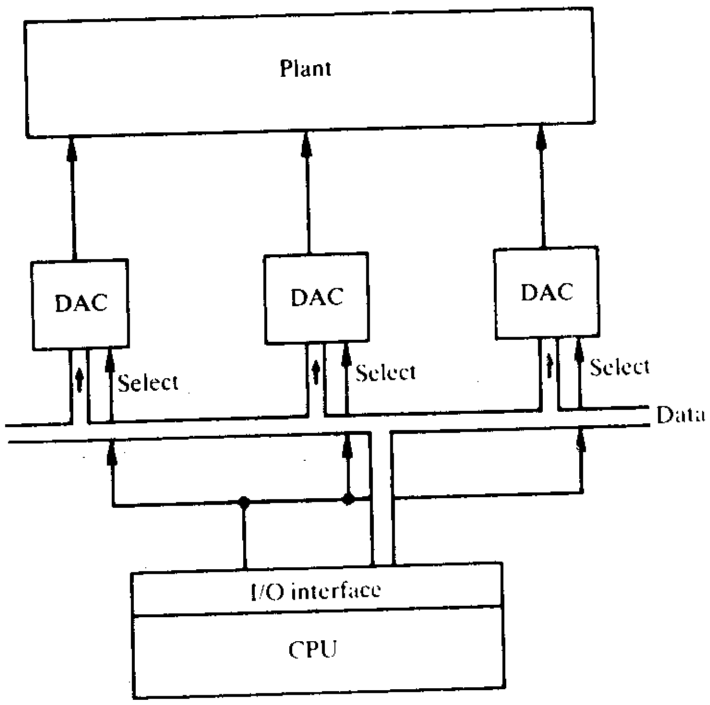

Figure 3.9 Analog output system.

3.5.4 Real-time Clock

A real-time clock is a vital auxiliary device for control computer systems. The hardware unit given the name 'real-time clock' may or may not be a clock; in many systems it is nothing more than a pulse generator with a precisely controlled frequency.

A common form of clock is based on using the ac supply line to generate pulses at 50 (or 60) times per second. By using slightly more complicated circuitry higher pulse rates can be generated, for example 100 (or 120) pulses per second. The pulses are used to generate interrupts and the interrupt handling software counts the interrupts and hence keeps time. If a greater precision in the time measurement than can be provided from the power supply is required then a hardware timer is used. A fixed frequency pulse generator (usually crystal-driven) decrements a counter which, when it reaches zero, generates an interrupt and reloads the count value. The interrupt activates the real-time clock software. The interval at which the timer generates an interrupt, and hence the precision of the clock, is controlled by the count value loaded into the hardware timer.

The choice of the basic clock interval, that is the clock precision, has to be a compromise between the timing accuracy required and the load on the CPU. If too small an interval is chosen, that is high precision, then the CPU will spend a large proportion of its time simply servicing the clock and will not be able to perform any other work.

The real-time clock based on the use of an interval timer and interrupt-driven software suffers from the disadvantage that the clock stops when the power is lost and on restart the current value of real time has to be entered. Real-time clocks are now becoming available in which the clock and date function are carried out as part of the interface unit, that is the unit acts like a digital watch. Real time can be read from the card and the card can be programmed to generate an interrupt at a specified frequency. These units are usually supplied with battery back-up so that even in the absence of mains power the clock function is not lost.

Real-time clocks are also used in batch processing and on-line computer systems. In the former, they are used to provide date and time on printouts and also for accounting purposes so that a user can be charged for the computer time used; the charge may vary depending on the time of day or day of the week. In on-line systems similar facilities to those of the batch computer system are required, but in addition the user expects the terminal to appear as if it is the only terminal connected to the system. The user may expect delays when the program is performing a large amount of calculation but not when it is communicating with the terminal. To avoid any one program causing delays to other programs, no program is allowed to run for more than a fraction of a second; typically timings are 200 ms or less. If further processing for a particular program is required it is only performed after all other programs have been given the opportunity to run. This technique is known as time slicing.

3.6 DATA TRANSFER TECHNIQUES

Although the meaning of the data transmitted by the various processes, the operator and computer peripherals differs, there are many common features which relate to the transfer of the data from the interface to the computer. A characteristic of most interface devices is that they operate synchronously with respect to the computer and that they operate at much lower speeds. Direct control of the interface devices by the computer is known as 'programmed transfer' and involves use of the CPU. Programmed transfer gives maximum flexibility of operation but because of the difference in operating speeds of the CPU and many interface devices it is inefficient. An alternative approach is to use direct memory access (DMA); the transfer requirements are set up using program control but the data transfers take place directly between the device and memory without disturbing the operation of the CPU (except that bus cycles are used).

With the reduction in cost of integrated circuits and microprocessors, detailed control of the input/output operations is being transferred to I/O processors which provide buffered entry. For a long time in on-line computing, buffers have been used to collect information (for example, a line) before invoking the program requesting the input. This approach is now being extended through the provision of I/O processors for real-time systems. For example, an I/O processor can be used to control the scanning of a number of analog input channels, only requesting main computer time when it has collected data from all the channels. This can be extended so that the I/O processor checks the data to test if any values are outside preset limits set by the main system.

A major problem in data transfer is *timing*. It may be thought that under programmed transfer, the computer can read or write at any time to a device, that is, can make an unconditional transfer. For some process output devices, for example switches and indicator lights connected to a digital output interface, or for DACs, unconditional transfer is possible since they are always ready to receive data. For other output devices, for example printers and communications channels, which are not fast enough to keep up with the computer but must accept a sequence of data items without missing any item, unconditional transfer cannot be used. The computer must be sure that the device is ready to accept the next item of data; hence either a timing loop to synchronise the computer to the external device or *conditional transfer* has to be used. Conditional transfer can be used for digital inputs but not usually for pulse inputs or analog inputs. Where unconditional transfer is used to read the digital value or an analog signal, or the value of a digital instrument rather than simply the pattern of logic indicators, then Gray code or some other form of cyclic binary code should be used to avoid the possibility of large transient errors.

3.6.1 Polling

A simple example of conditional transfer is shown in Figure 3.10. Assuming that the data is being transferred to a printer which operates at 40 characters per second, the computer will find that the device is ready once every 25 milliseconds. The three instructions involved in performing the test will take approximately 5 μs (the actual time will depend on the speed of the processor); thus the conditional test will be carried out 5000 times for each character transmitted. The computer will spend 99.98% of its time in checking to see if the device is ready and only 0.02% of the time doing useful work; this is clearly inefficient.

A software timing loop can be used as an alternative to a status line on the interface. For example, a delay can be created by loading a register with a number and repeatedly decrementing the register until it reads zero:

```
     LD B, 25     ;load register B with time delay
LOOP DEC B        ;decrement B
     JR NZ, LOOP  ;repeat until B is zero
```

To ensure that no transfer is made before the peripheral is ready the time delay must be slightly greater than the maximum delay expected in the peripheral; thus

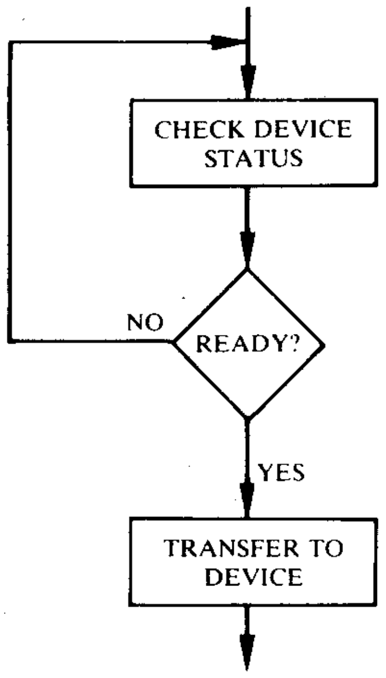

Figure 3.10 Conditional transfer (busy wait).

in terms of use of the CPU this method is even more inefficient than the use of the conditional wait. However, it does slightly simplify and reduce the cost of the interface.

An alternative arrangement for conditional transfer, which allows the computer to continue doing useful work if the device is busy, is shown in Figure 3.11. In this method a check is made to see if the device is ready: if it is ready then the transfer is made; if it is not the computer continues with other work and returns at some later time to check the device again. The technique avoids the inefficiency of waiting in a loop for a device to become ready, but presents the programmer with the difficult task of arranging the software such that all devices are checked at frequent intervals.

Conditional transfer techniques involve *polling*, which is using the computer to check whether a device is ready for a data transfer. The problems of polling using conditional waits can be avoided if the computer can respond to an interrupt signal.

3.6.2 Interrupts

An interrupt is a mechanism by which the flow of the program can be temporarily stopped to allow a special piece of software – an interrupt service routine (also called an interrupt handler) – to run. When this routine has finished, the

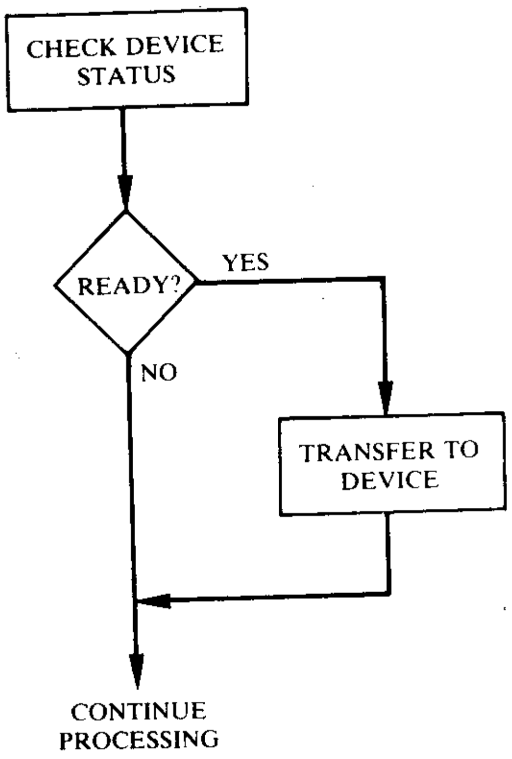

Figure 3.11 Conditional transfer.

program which was temporarily suspended is resumed. The process is illustrated in Figure 3.12.

Interrupts are essential for the correct operation of most real-time computer systems; in addition to providing a solution to the conditional wait problem they are used for:

Real-time clock: The external hardware provides a signal at regularly spaced intervals of time; the interrupt service routine counts the signals and keeps a clock.

Alarm inputs: Various sensors can be used to provide a change in a logic level in the event of an alarm. Since alarms should be infrequent, but may need rapid response times, the use of an interrupt provides an effective and efficient solution.

Manual override: Use of an interrupt can allow external control of a system to allow for maintenance and repair.

Hardware failure indication: Failure of external hardware or of interface units can be signalled to the processor through the use of an interrupt.

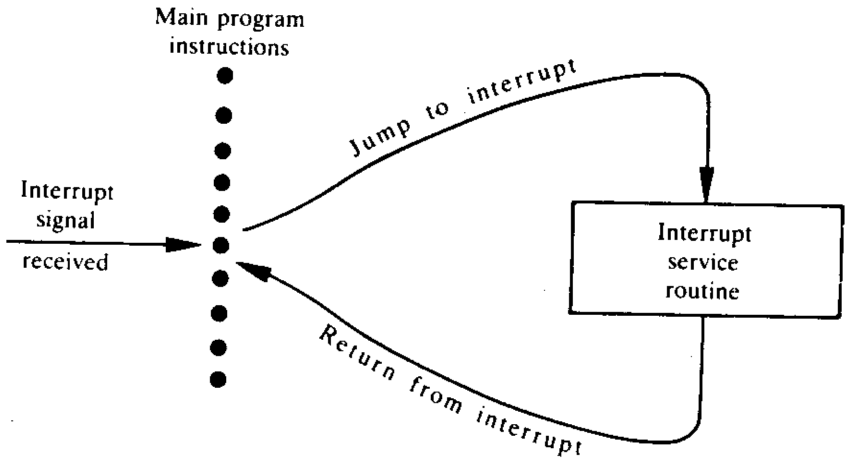
Main program instructions
Jump to interrupt
Interrupt signal received
Interrupt service routine
Return from interrupt

Figure 3.12 Interrupt-driven program control transfer.

Debugging aids: Interrupts are frequently used to insert breakpoints or traces in the program during program testing.

Operating system: Interrupts are used to force entry to the operating system before the end of a time slice.

Power failure warning: It is simple to include in the computer system a circuit that detects very quickly the loss of power in the system and provides a few milliseconds' warning before the loss is such that the system stops working. If this circuit is connected to an interrupt which takes precedence over all other operations in the computer there can be sufficient time to carry out a few instructions which could be sufficient to close the system down in an orderly fashion.

3.6.2.1 Saving and restoring registers

Since an interrupt can occur at any point in a program, precautions have to be taken to prevent information which is being held temporarily in the CPU registers from being overwritten. All CPUs automatically save the contents of the program counter; this is vital. If the contents were not saved then a return to the point in the program at which the interrupt occurred could not be made. Some CPUs, however, do more and save all the registers. The methods commonly used are:

- Store the contents of the registers in a specified area of memory. (Note that this implies that an interrupt cannot be interrupted – see below.)
- Store the registers on the memory stack. This is a simple, widely used method which permits multi-level interrupts; the major disadvantage is the danger of stack overflow.

- Use of an auxiliary set of registers. Some processors provide two sets of the main registers and an interrupt routine can switch to the alternative set. If only two sets are provided multi-level interrupts cannot be handled. An alternative method is to use a designated area of memory as the working registers and then an interrupt only requires a pointer to be changed to change the working register set.

The use of automatic storage of the working registers is an efficient method if all registers are to be used; it is inefficient if only one or two will be used by the interrupt routine. For this reason fully automatic saving is usually restricted to CPUs with only a few working registers in the CPU; systems with many working registers provide an option either to save or not to save. Unless response time is critical it is good engineering practice to save all registers: in this way there is no danger, in a subsequent modification to the interrupt service routine, of using a register which is not being saved, and failing to add it to the list of registers to be saved. The resulting error would be difficult to find since it would cause random malfunctioning of the system.

The machine status must of course be restored on exit from the interrupt routine; this is straightforward for all methods except that which uses the stack to save registers; in this case the registers are restored in the opposite order than that in which they were saved. Systems providing automatic saving also provide automatic restore on exit from the interrupt.

An example of the framework of an interrupt service routine is shown below.

```
INT1: CALL SAVREG  ;SAVREG is routine which saves
                   ;working registers
;
;code for interrupt handling is inserted here
;
      CALL RESREG  ;RESREG is routine which
                   ;restores working registers
      EI           ;enable interrupts
      RETI         ;return from interrupt routine
```

The above routine is suitable for a system in which interrupts are not allowed to be interrupted; hence the EI instruction which enables interrupts is not executed until immediately prior to the return from interrupt. The return from an interrupt routine has to be handled with care to prevent unwanted effects. For example, the EI instruction does not re-enable interrupts until after the execution of the instruction which immediately follows it. Therefore, by using the EI/RETI combination a pending interrupt cannot take effect until after the return from the previous one has been completed. In this example it is assumed that the microprocessor automatically disables interrupts on acknowledgement of an interrupt and they remain disabled until an EI instruction is executed. Some CPUs operate by disabling interrupts for only one instruction following an interrupt

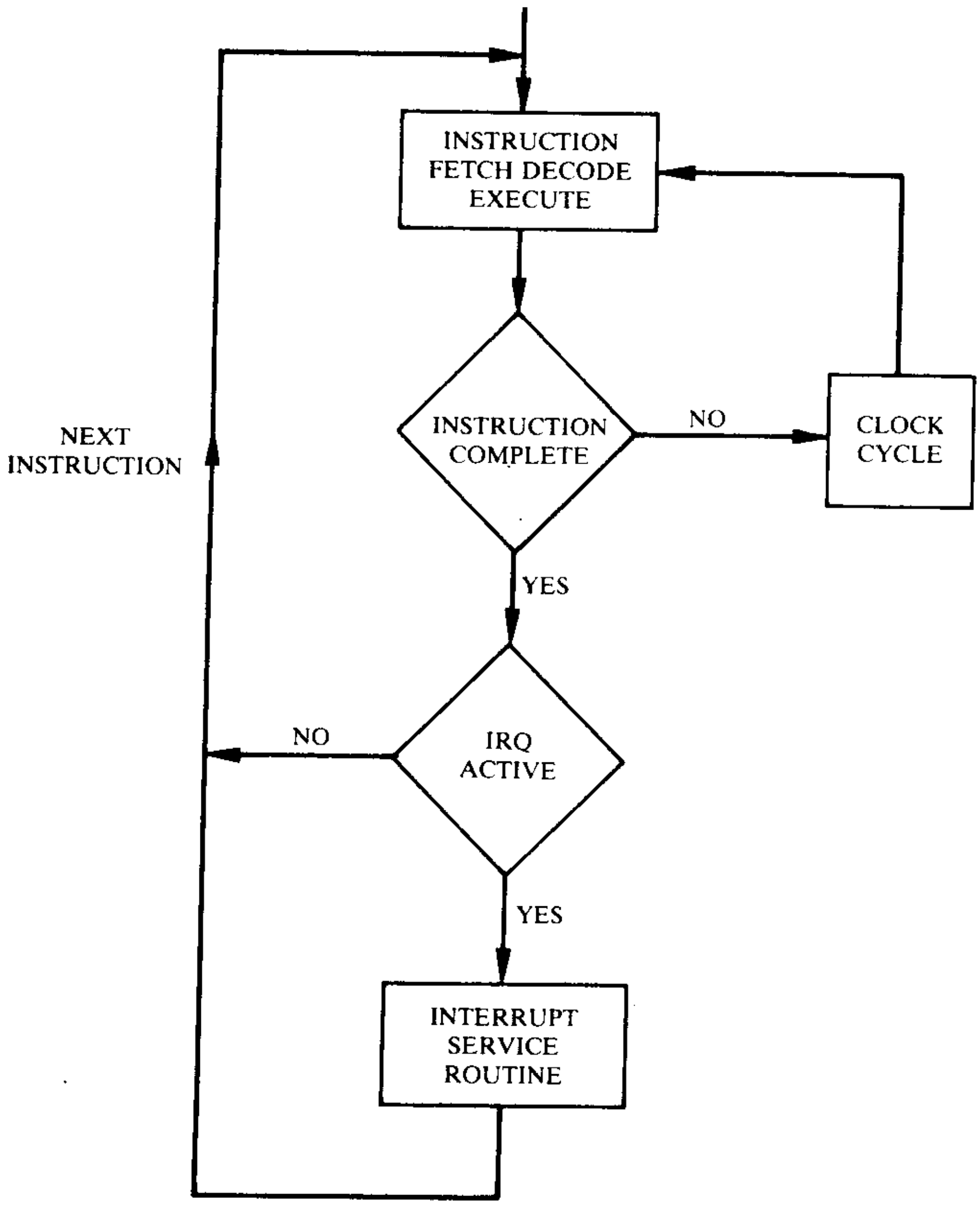

Figure 3.13 Flowchart of basic interrupt mechanism.

acknowledge; it thus becomes the responsibility of the programmer to disable interrupts as the first instruction of the interrupt service routine if only a single level of interrupt is to be supported.

3.6.2.2 Interrupt input mechanisms

A simple form of interrupt input is shown in Figure 3.13. In between each instruction the CPU checks the IRQ line. If it is active, an interrupt is present and the interrupt service routine is entered; if it is not active the next instruction is fetched and the cycle repeats. Note that an instruction involves more than one CPU clock cycle and that the interrupt line is checked only between instructions. Because several clock cycles may elapse between successive checks of the interrupt line, the interrupt signal must be latched and only cleared when the interrupt is acknowledged.

A common arrangement is to have two interrupt lines as shown in Figure 3.14: one of the lines, IRQ, can be enabled and disabled using software and hence the computer can run in a mode in which external events cannot disturb the processing. A second interrupt line is provided; this interrupt cannot be turned off by software and hence it is said to be a non-maskable interrupt (NMI). A typical use would be to provide the power failure detect interrupt.

Although most modern computer CPUs have only one or two interrupt lines, a large number of interrupts can be connected by means of an OR gate. It then

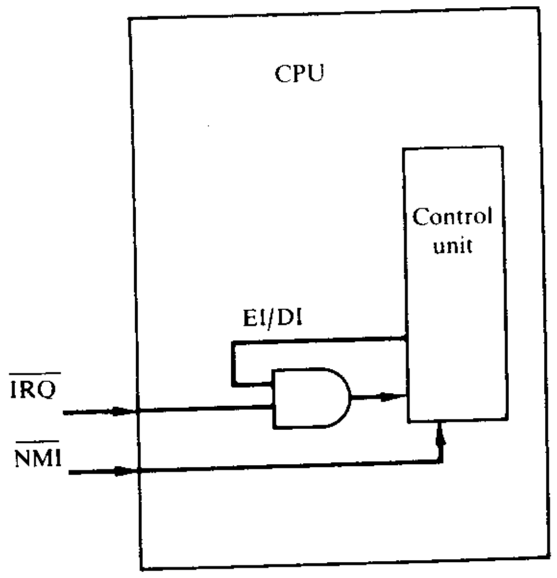

Figure 3.14 Typical basic interrupt system.

becomes a problem to determine which of the many external interrupt lines has generated the CPU interrupt.

3.6.2.3 Interrupt response mechanisms

The CPU may respond to the interrupt in a variety of ways; some of the more popular methods are given below:

1. Transfer control to a specified address – usually in the form of a 'call' instruction.
2. Load the program counter with a new value from a specified register or memory location.
3. Execute a 'call' instruction but to an address supplied from the external system.
4. Use an output signal – an Interrupt Acknowledge – to fetch an instruction from an external device.

Methods 1 and 2 are said to be *software biased*, in that they require little in the way of external hardware and rely instead on software to determine the interrupt source and the appropriate interrupt service routine. Methods 3 and 4 are *hardware biased* in that they require more external hardware but can identify the interrupt source and can transfer program control directly to the appropriate interrupt service routine.

In method 2 the address of the interrupt response routine is stored in specified memory locations; the address stored here is called the *interrupt vector* or *interrupt response vector*. Once the interrupt is detected control is passed to an interrupt response routine and polling must be used to determine which device has caused the interrupt.

The use of polling in interrupt systems has the advantage over normal polling systems that at least one of the inputs is guaranteed to be active. It is clearly, however, not a very satisfactory system if large numbers of devices have to be checked. The load can be reduced by testing the devices which interrupt most frequently first, but this may conflict with response time requirements in that a device which interrupts infrequently may require a rapid response time and hence should be checked first. If an equal response time, on average, for each device is required it will be necessary to rotate the order in which the devices are checked. Doing this can also prevent one device which interrupts frequently from locking out all others. The method does provide a flexible way of allocating priority to the various devices which can generate interrupts.

3.6.2.4 Hardware vectored interrupts

Methods 3 and 4 require the use of some form of vectored interrupt structure to identify which of the external devices has generated an interrupt. They also require a mechanism to arbitrate between the possible sources of the interrupt to prevent more than one interrupt activating the IRQ at any one time. The process of arbitration involves assigning priorities to the various interrupts.

A frequently used arrangement is the daisy chain in which an 'acknowledge' signal is propagated through the devices until it is blocked by the interrupting device. Figure 3.15 shows a typical arrangement. Each unit has an IEI (Interrupt Enable In) pin and an IEO (Interrupt Enable Out) pin; it is assumed that on both pins the active signal is high. The first IEI in the chain is set permanently on 'high'. For any given

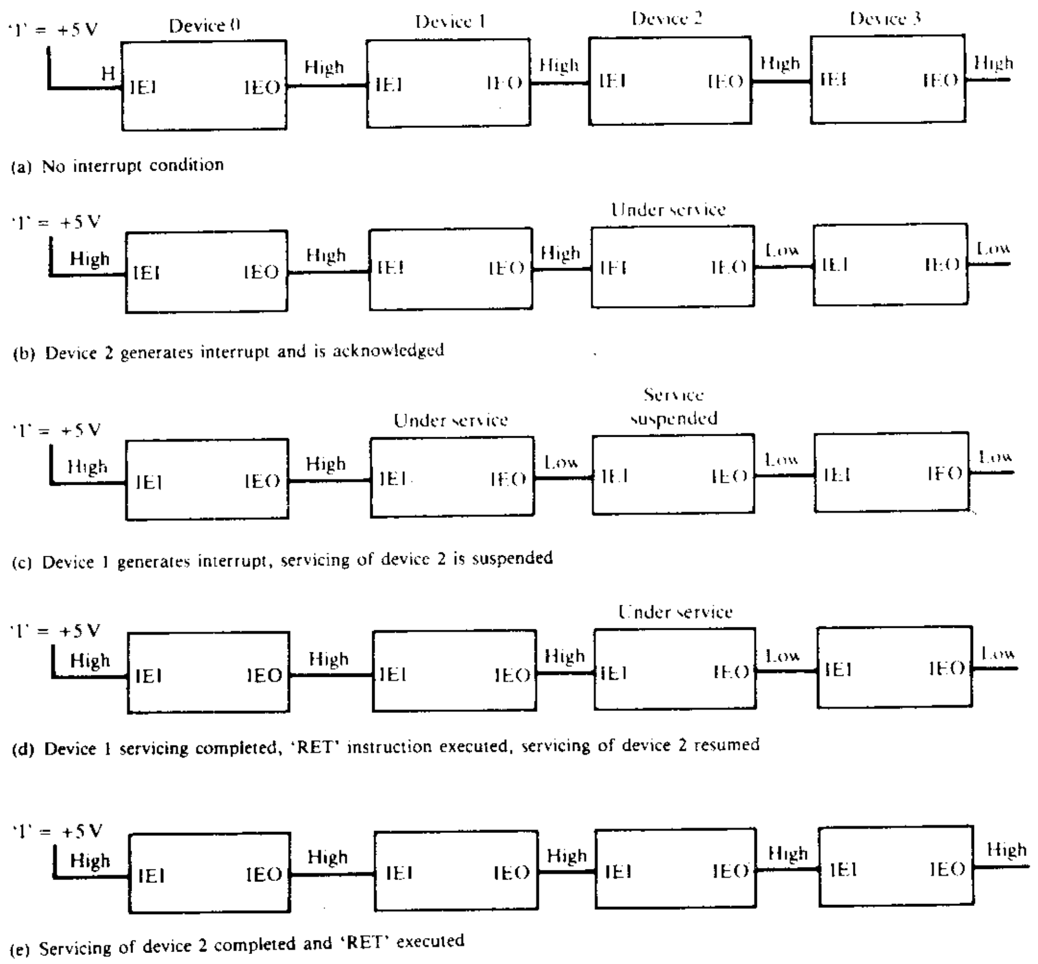
'1' = +5 V
Device 0
Device 1
Device 2
Device 3
H
IEI
IEO
High
(a) No interrupt condition
Under service
Low
(b) Device 2 generates interrupt and is acknowledged
Service suspended
(c) Device 1 generates interrupt, servicing of device 2 is suspended
(d) Device 1 servicing completed, 'RET' instruction executed, servicing of device 2 resumed
(e) Servicing of device 2 completed and 'RET' executed

Figure 3.15 Daisy-chain interrupt structure: (a) no interrupt condition; (b) device 2 generates interrupt and is acknowledged; (c) device 1 generates interrupt, servicing of device 2 is suspended; (d) device 1 servicing completed, 'RET' instruction executed, servicing of device 2 resumed; (e) servicing of device 2 completed and 'RET' executed.

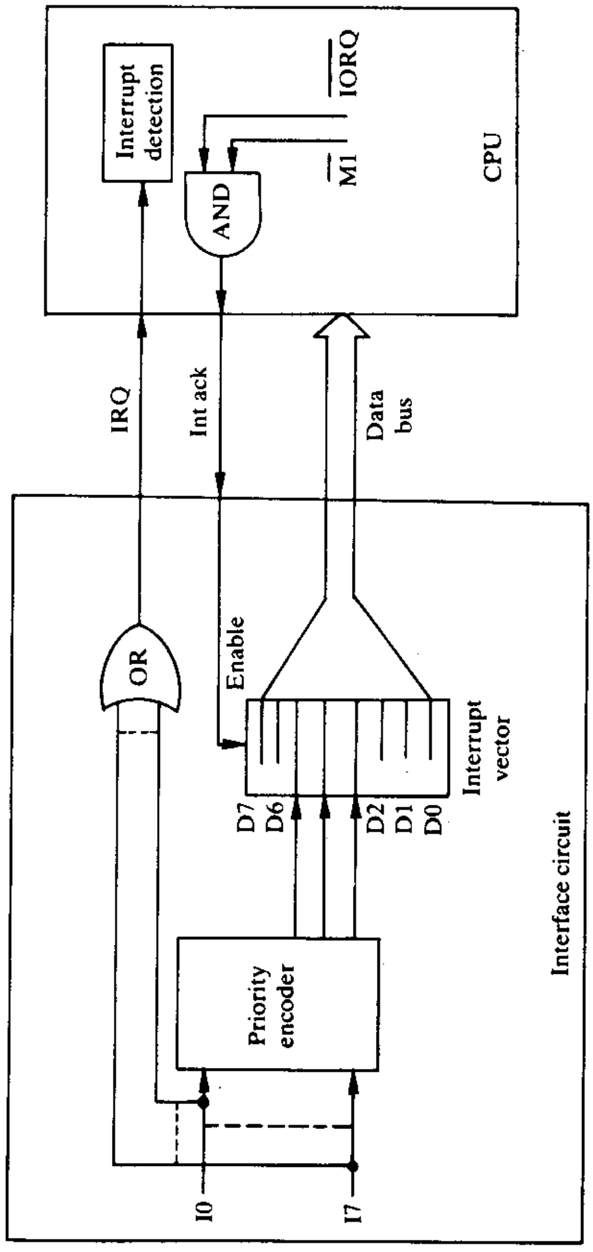

Figure 3.16 Interrupt vectoring using priority encoding circuit.

unit the output pin IEO is high if, and only if, the input IEI is high and the unit is not requesting an interrupt. If a device is requesting an interrupt and IEI is high, that device should set IEO low and in response to an 'interrupt acknowledge' signal send its interrupt vector. If a device is requesting an interrupt but the IEI is low then it should not respond to the interrupt acknowledge signal.

In Figure 3.15a the system is shown with no interrupts active, so all the signals are at high. The effect of Device 2 generating an interrupt and it being acknowledged is shown in Figure 3.15b: Device 2 is serviced, Device 3 is locked out and Devices 0 and 1 can still interrupt. If Device 1 now generates an interrupt the servicing of Device 2 will be suspended in favour of Device 1 (see Figure 3.15c). On completion of the servicing of Device 1, the servicing of Device 2 resumes (Figure 3.15d). Finally, when this is completed, all the IEI/IEO signals return to 'high' and all devices are enabled. The assumption implicit in this arrangement is that one interrupt service routine can itself be interrupted by a higher-priority interrupt. This is known as a multi-level interrupt structure and is dealt with below.

In a daisy-chain arrangement the device priority is determined by the position of the device in the chain and cannot be changed by the software. Great care has to be given to timing considerations; in particular, care has to be taken to allow the IEI signal time to propagate along the chain.

The determination of interrupt priority can be performed using priority encoder circuits (see Figure 3.16). In this system an interrupt occurring on any line causes the interrupt line (IRQ) to become active and also places a three-bit code specifying the number of the interrupt line which is active on the data bus. In the event of more than one line being active the priority encoder supplies the number of the highest-priority interrupt – the usual arrangement is that the line with the lowest number is considered to be of highest priority.

3.6.2.5 Interrupt response vector

The interrupt response vector in the above systems can take a variety of forms: it may be an instruction, the address of the interrupt service routine, the address of a pointer to an interrupt service routine, or part of the address of the interrupt service routine or pointer.

A widely used method is to employ an interrupt mechanism in which the interrupting device supplies the address of the location in which the pointer to the start of the interrupt routine is stored. When a device interrupts it supplies this address and the CPU loads the program counter with the contents of the interrupt vector location and hence control of the CPU passes to the first instruction of the interrupt service routine.

3.6.2.6 Multi-level interrupts

In most real-time systems a single interrupt level is unacceptable; the whole purpose of interrupts is to get a fast response and this would be prevented if a low-priority interrupt could lock out a high-priority one. A typical picture of multi-level interrupts is shown in Figure 3.17. An application program (the main task) is interrupted at regular intervals by the clock interrupt which is the highest-priority interrupt (level 0). When the interrupt occurs control is passed to the clock interrupt service routine (ISR 0) – transfers 1, 2 and 3 in Figure 3.17. During the servicing of the clock interrupt, the printer generates an interrupt request (4), but since the printer is of lower priority than the clock the interrupt is not dealt with until the clock routine ISR 0 has finished. When this occurs, instead of control returning to the main program it passes to the printer service routine ISR 1 (5). The printer service routine does not complete before the next clock interrupt, so it is suspended (6) while the next interrupt from the clock is dealt with. At the termination of the clock routine return is made to the printer (7) and finally, when the printer ISR finishes, a return is made to the main program (8).

It should be obvious that the ability to interrupt an interrupt service routine should be restricted to interrupts which are of higher priority than the routine executing. In order to do this there has to be some facility for masking out (or inhibiting) interrupts of lower priority. Masking is achieved automatically in the daisy-chain system, since a device which wishes to interrupt lowers its IEO line thus preventing all lower-priority devices from responding to the interrupt acknowledge signal. In the daisy-chain system, however, the device must receive a signal from the

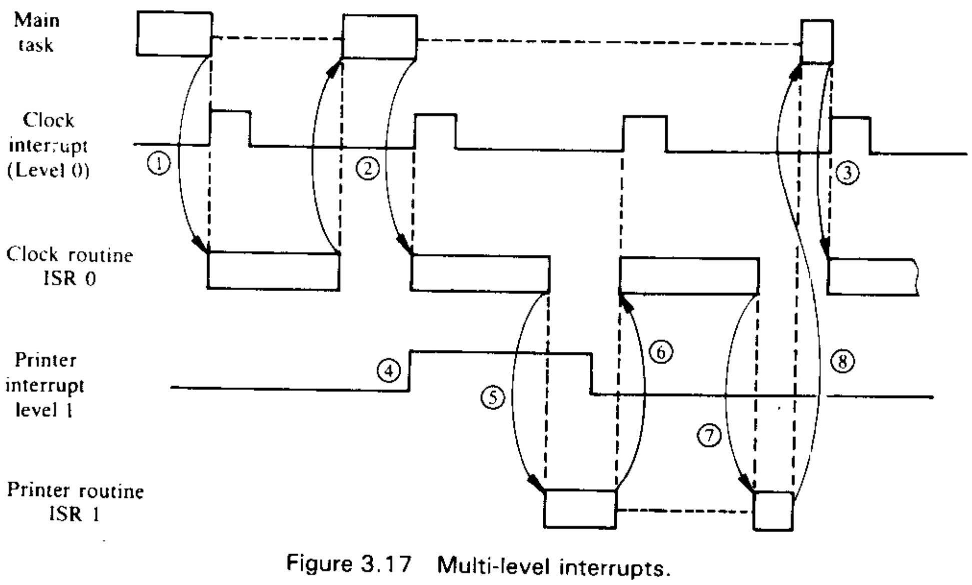

Figure 3.17 Multi-level interrupts.

CPU on return from the interrupt service routine in order that it can set the IEO line high and hence permit access to the system by lower-priority devices.

An alternative scheme used is to have a mask register which can be loaded from software and used to inhibit the lower-priority interrupt lines. Figure 3.18 shows a system which uses a priority encoder and the software sequence is outlined in Figure 3.19. Note that with the mask system it is possible to mask out any interrupt, not just ones with lower priority. This can have advantages if, for example, a high-priority alarm interrupt is continually being generated because of a fault on the plant; once the fault condition has been recognised it is desirable to mask out the interrupt to avoid the computer spending all its time simply servicing the interrupt. The ability to mask out selected levels provides software-controlled priority reallocation.

Figure 3.19 shows typical functions performed by an interrupt service routine. The first requirement is to save the working environment; the current mask register must also be saved and then the new mask register sent out. The interrupts can now be enabled and the actual servicing of the interrupt commenced. When the servicing is completed the interrupts are disabled, the previous mask register restored and the working environment restored. The interrupts can now be enabled and a return from

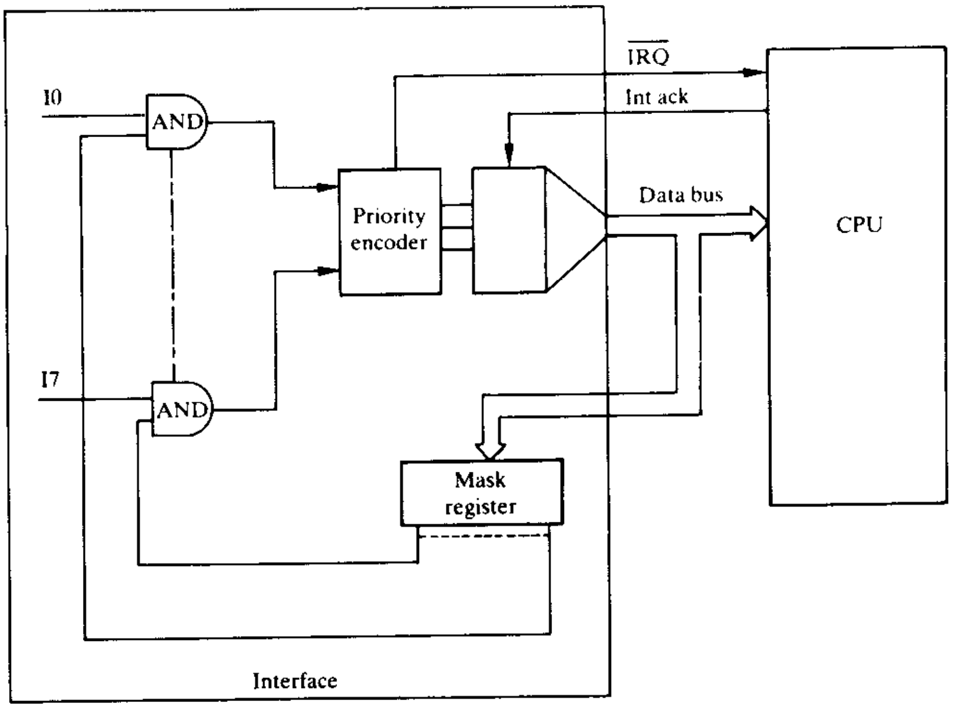

Figure 3.18 Interrupt masking.

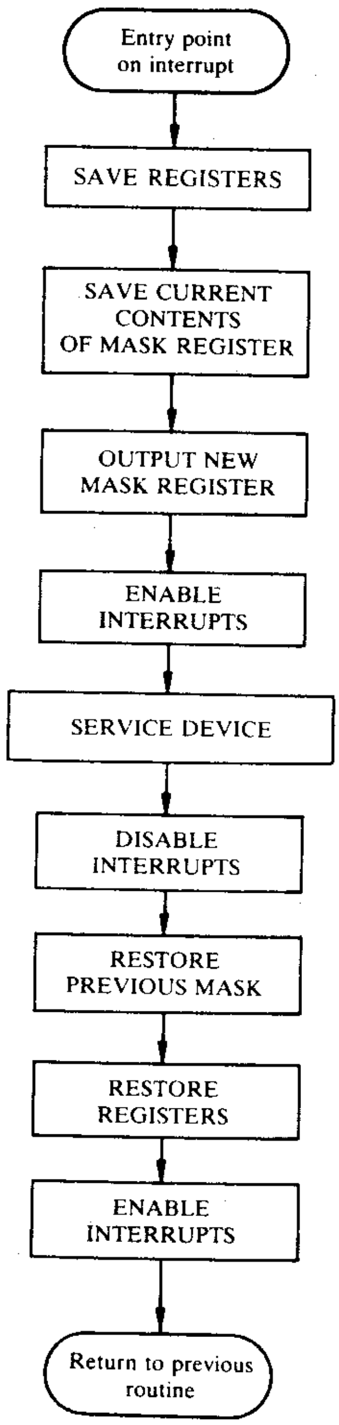

Figure 3.19 Function performed by interrupt servicing routine.

interrupt executed. Note that some computer systems automatically disable all lower-priority interrupts and hence the need to save and restore mask registers is avoided.

3.6.3 Direct Memory Access

Three methods are normally used – burst mode, distributed mode and cycle stealing. In burst mode, the DMA controller takes over the data highways of the computer and locks out the CPU for the period of time necessary to transfer, say, 256 bytes between fast memory and backing memory. The use of burst mode can seriously affect the response time of a real-time system to an external event and because of this may not be acceptable.

In distributed mode the DMA controller takes occasional machine cycles from the CPU's control and uses each cycle to transfer a byte of information between fast memory and backing memory. In a non-real-time system the loss of these machine cycles to the CPU is not noticeable. However, in a real-time system which uses software timing loops the loss of machine cycles will then affect the time taken to complete the timing loop. The program is unaware of the machine cycles used by the DMA controller and hence will still cycle through the same number of instructions; however, the elapsed time may be the equivalent of 200 machine cycles rather than the expected 100 cycles.

The cycle-stealing method only transfers data during cycles when the CPU is not using the data bus. Therefore the program proceeds at the normal rate and is completely unaffected by the DMA data transfers. This is, however, the slowest method of transfer between fast memory and backing store.

3.6.4 Comparison of Data Transfer Techniques

Polling, with either busy wait or periodic checks on device status, provides the simplest method of data transfer, in terms of the programming requirements and in the testing of programs. The use of interrupts results in software which is much less structured than a program with explicit transfers of control; there are potential transfers of control at every point in the program.

Interrupt-driven systems are much more difficult to test since many of the errors may be time dependent. A simple rule is to check the interrupt part of the program if irregular errors are occurring. The generation of appropriate test routines for interrupt systems is difficult; for proper testing it is necessary to generate random interrupt patterns and to carry out detailed analysis of the results.

At high data transfer rates the use of interrupts is inefficient because of the overheads involved in the interrupt service routine – saving and restoring the environment – hence polling is often used. An alternative for high rates of transfer is to substitute hardware for software control and use direct memory access techniques.

3.7 COMMUNICATIONS

The use of distributed computer systems implies the need for communication: between instruments on the plant and the low-level computers (see Figure 3.20); between the Level 1 and Level 2 computers; and between the Level 2 and the higher-level computers. At the plant level communications systems typically involve parallel analog and digital signal transmission techniques since the distances over which communication is required are small and high-speed communication is usually required. At the higher levels it is more usual to use serial communication methods since, as communication distances extend beyond a few hundred yards, the use of parallel cabling rapidly becomes cumbersome and costly.

As the distance between the source and receiver increases it becomes more difficult, when using analog techniques, to obtain a high signal-to-noise ratio; this is particularly so in an industrial environment where there may be numerous sources of interference. Analog systems are therefore generally limited to short distances. The use of parallel digital transmission provides high data transfer rates but is expensive in terms of cabling and interface circuitry and again is normally only used over short distances (or when very high rates of transfer are required).

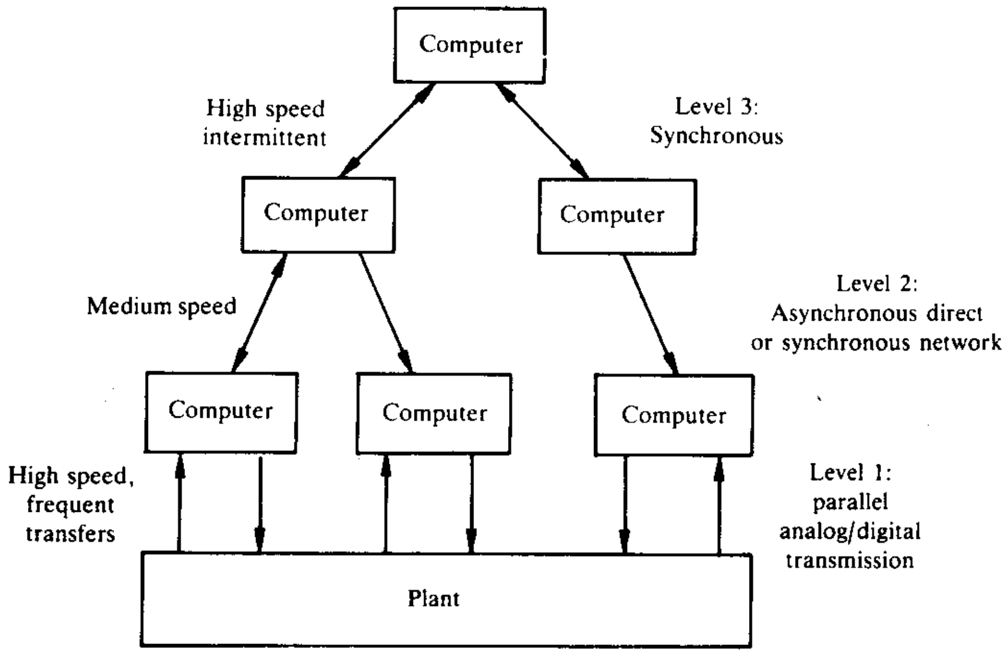
Computer
High speed intermittent
Level 3: Synchronous
Computer
Computer
Medium speed
Level 2: Asynchronous direct or synchronous network
Computer
Computer
Computer
High speed, frequent transfers
Level 1: parallel analog/digital transmission
Plant

Figure 3.20 Data transmission links.

Serial communication techniques can be characterised in several ways:

1. Mode
   (a) asynchronous
   (b) synchronous
2. Quantity
   (a) character by character
   (b) block
3. Distance
   (a) local
   (b) remote, that is wide area
4. Code
   (a) ASCII
   (b) other

3.7.1 Asynchronous and Synchronous Transmission Techniques

Asynchronous transmission implies that both the transmitter and receiver circuits use their own local clock signals to gate data on and off the data transmission line. So that the data can be interpreted unambiguously there must be some agreement between the transmitter and receiver clock signals. This agreement is forced by the transmitter periodically sending synchronisation information down the transmission line. An alternative approach is to use an additional physical connection – a clock wire – and periodically to send a synchronising signal.

The most common form of asynchronous transmission is the character-by-character system which is frequently used for connecting terminals to computer equipment and was introduced for the transmission of information over telegraph lines. It is sometimes called the stop–start system. In this system each character transmitted is preceded by a 'start' bit and followed by one or two 'stop' bits (see Figure 3.21). The start bit is used by the receiver to synchronise its clock with the incoming data; for correct transfer of data the clock and data signals must remain synchronised for the time taken to receive the following eight data bits and two stop bits. The transmission is thus *bit* synchronous but *character* asynchronous. The advantage of the stop–start system is that, particularly at the lower transmission rates, the frequencies of the clock signal generators do not have to be closely matched. The disadvantage is that for each character transmitted (seven bits) three or four extra bits of information have also to be transmitted and thus the overall information ratio is not very high.

To overcome the problem of transmitting redundant bits, synchronous systems designed to transmit large volumes of data over short periods of time use block-synchronous transmission techniques. The characters are grouped into records, for example blocks of 80 characters, and each record is preceded by a synchronisation signal and terminated with a stop sequence. The synchronisation sequence is used to enable the receiver to synchronise with the transmitter clock.

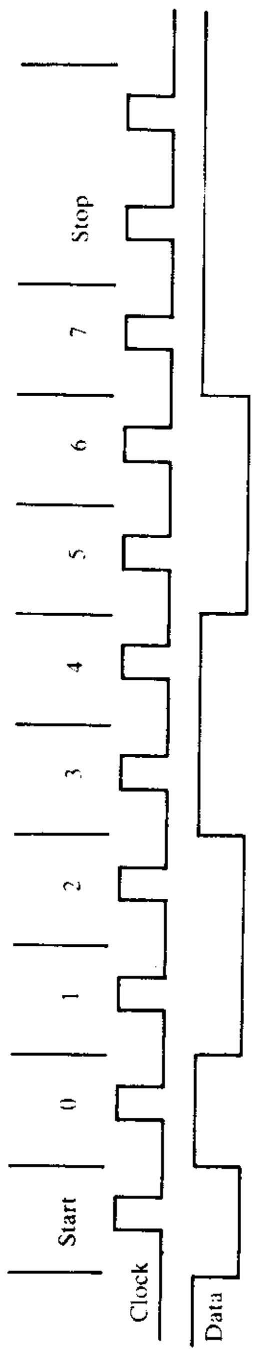

Figure 3.21 Asynchronous transmission.

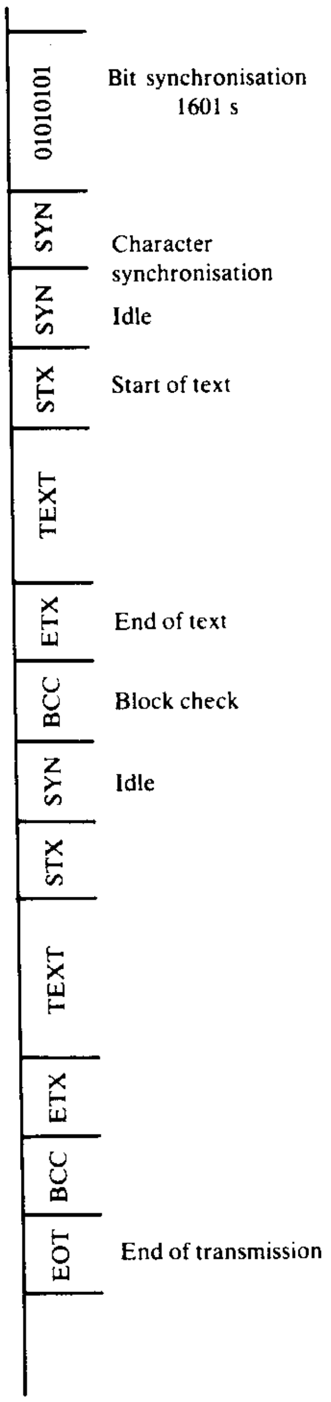

Figure 3.22 Synchronous transmission.

To establish effective communication more than just a synchronisation signal must be transmitted – the additional information is called the *protocol*. A simple protocol is shown in Figure 3.22. At the start of a transmission, bit synchronisation is achieved by the transmitter sending out a sequence of zeros and ones, followed by the ASCII code 'SYN'. The transmitter will continue to send the SYN code until the receiver responds by sending back the code 'ACK' or a preset time elapses (device time out); if time out occurs, the transmitter sends the bit pattern of zeros and ones again. Once contact has been established the transmitter will send out SYN characters during any idle period and the receiver will respond by sending back ACK; the line will only be completely idle when an EOT (End Of Transmission) character has been sent by the transmitter. The text is broken up into blocks and each block is preceded by an STX (Start of TeXt) character and ended by an ETX (End of TeXt) character. Following the ETX will be an integrity check on the data; typically this will take the form of a parity check.

There are two main standards for synchronous transmission systems:

1. BISYNC (BInary SYNchronous Communication). This is the older system used in IBM equipment and is obsolescent.
2. HDLC (High-level Data Link Control). This is used in most new equipment.

In synchronous transmission systems the clock signal for the timing of the data transfer is provided solely by the transmitter and is sent to the receiver even when no data is being transmitted. When data is transmitted it is superimposed on the clock signal. With synchronous transmission there is no need to transmit extra bits to enable the receiver clock to synchronise with the transmitter and hence the effective data transmission rate for a given speed of line is higher. The disadvantage is that the interface circuitry is more complex and hence more expensive. The use of synchronous transmission does not avoid the need for a transmission protocol. The advantage of block transmission is that a much higher ratio of data bits to control bits can be obtained.

3.7.2 Local- and Wide-area Networks

Wide-area networks have existed for many years and they operate over a very wide geographical area (many are international networks) at moderate speeds. The local-area network (LAN) is a more recent development and it is having a considerable impact on the design of process control equipment. LANs make use of a wide range of transmission media such as twisted pair, co-axial cable, and fibre optics; they operate at a range of transmission speeds (up to 240 Mbit s^{-1}) and use a range of different protocols and topologies.

Typical topologies are shown in Figure 3.23. For computer control applications no one topology represents the best solution: the particular application will

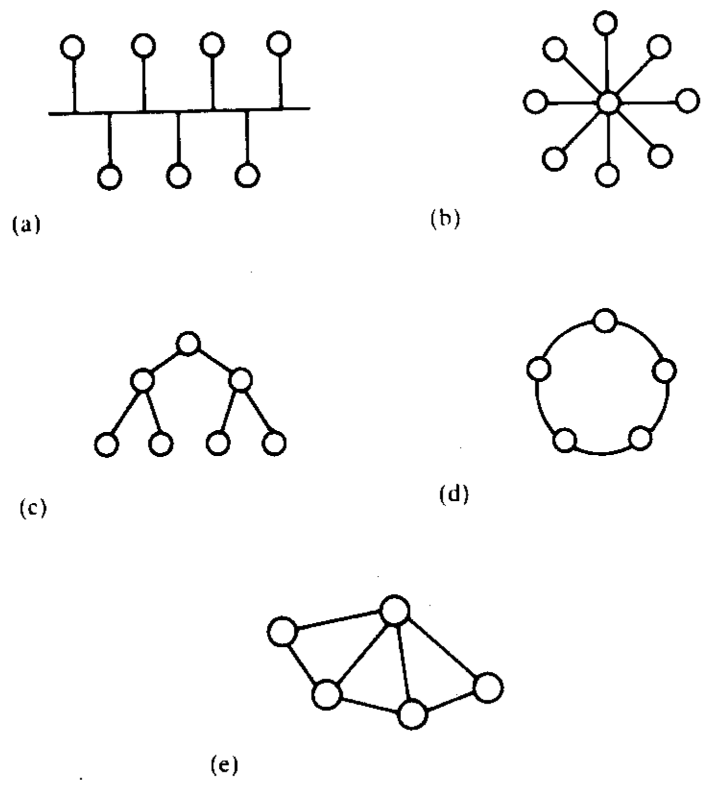

Figure 3.23 LAN topologies: (a) data bus; (b) star; (c) hierarchy (or tree); (d) ring; (e) mesh.

govern the most appropriate one. The characteristics of each are briefly outlined below.

Data bus: This is the simplest of all the LAN topologies. The bus is normally passive and all the devices are simply plugged into the transmitting medium. The bus is inherently reliable because of its passive nature but there may be a limitation on the length of a bus in that any transmitting device connected has to be able to transmit for the full length of the bus. It is a broadcast system and hence a packet of data placed on the bus is available to all devices.

Star: The star network is not very widely used; it depends on a central switching node to which all other nodes are connected by a bidirectional link. Data sent to the central switch can be forwarded either in the broadcast mode, that is to all other nodes, or only to a specified node. The computer-controlled PABX (Private Automatic Branch eXchanges) used in many businesses operate in the star mode – all the telephone lines connect to the central unit – while the forwarding system used is to a specified node. A weakness of the star topology is that the central node is a critical component in the system; if it fails, the whole system fails.

Hierarchy: The system has many of the characteristics of the star, but instead of one central switching node, many of the nodes have to act as switches. Frequently it can closely reflect the actual structure of the application. The addition of new nodes to a hierarchy can be difficult.

Ring: This is probably the most popular method. The ring is typically an active transmission system, that is the ring itself contains regeneration circuits which amplify the signals. The information placed on a ring network continues to circulate until a device removes it from the ring; in some systems the originating device removes the data from the ring. The information is broadcast in the sense that it is available to all devices connected to the ring.

Mesh: The mesh topology allows for random interconnection between the various nodes. It provides a means by which alternative routes between nodes can be found and hence has built into it a form of redundancy. A problem which can arise with the mesh is that there can be a variable delay between the sending and receiving of the message because of the number of nodes through which the message has had to pass. Information is transmitted in the form of 'packets' which may be of fixed or variable length. Early systems used character-oriented packets similar to that illustrated in Figure 3.22. The prevailing standard is now the HDLC protocol (referred to in the previous section) and the format of the packet is shown in Figure 3.24.

The *access* mechanisms used to ensure that only one node of the network is attempting to transmit at any one time divide into two main types:

- synchronous token passing, message slots; and
- asynchronous carrier sense multiple access/collision detection (CSMA/CD).

Ring-based LANs normally use synchronous techniques. The packets of data circulate in one direction around the ring and in the token passing system attached to one, and only one, packet is a token. (If the network is idle a packet containing just the token is circulated.) Each node reads the packet into a buffer and checks the message. There are three actions which can then be taken.

1. If the message is for that node it is read, marked as accepted and replaced

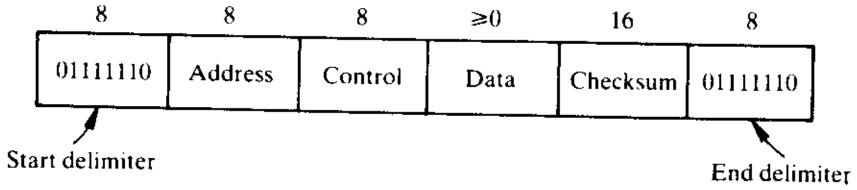

Figure 3.24 Format of HDLC packet.

on the ring – it will be removed from the ring when it reaches the originating node.
2. If the message is not for that node it is simply replaced on the ring.
3. If the packet contains the token and the node wishes to transmit a message, the token is removed from the packet which is then passed on. The node then transmits its own message adding the token to the end of the message.

An alternative to the token passing method is the message slot. A sequence of bits is used to mark a slot and the slots circulate around the ring. If a node detects an empty slot it may insert a message in that slot.

Bus LANs may use token passing (the token is passed from node to node in some predetermined manner) or message slots but asynchronous methods are more common. In the asynchronous systems a node may attempt to transmit at any time. The node listens to the bus and if it is idle begins to transmit. Because of the distances between nodes and the time taken to transmit a message, two (or more) nodes may be transmitting simultaneously. If this happens a collision is said to have occurred; bus systems must therefore have some means of detecting a collision. If a collision is detected the nodes attempting to transmit execute a random delay and then retry.

Several of the major process companies have developed distributed control systems based on the use of LAN technology (mostly rings). These allow a wide range of devices – from individual instruments to large computers – to be connected to a common network.

3.8 STANDARD INTERFACES

Most of the companies which supply computers for real-time control have developed their own 'standard' interfaces, such as the Digital Equipment Corporation's Q-bus for the PDP-11 series, and, typically, they, and independent suppliers, will be able to offer a large range of interface cards for such systems. The difficulty with the standards supported by particular manufacturers is that they are not compatible with each other; hence a change of computer necessitates a redesign of the interface.

An early attempt to produce an independent standard was made by the British Standards Institution (BS 4421, 1969). Unfortunately the standard is limited to the concept of how the devices should interconnect and the standard does not define the hardware. It is not widely used and has been overtaken by more recent developments.

An interface which was originally designed for use in atomic energy research laboratories – the computer automated measurement and control (CAMAC) system – has been widely adopted in laboratories, the nuclear industry and some other industries. There are over 1000 different modules, supported by about 50 manufacturers in eight countries, available for the system. There are also FORTRAN libraries which provide software to support a wide range of the interface modules. One of the attractions of the system is that the CAMAC data highway connects to the computer by a special card; to change to a different computer only requires that the one card be changed.

A general purpose interface bus (GPIB) was developed by the Hewlett Packard Company in the early 1970s for connecting laboratory instruments to a computer. The system was adopted by the IEEE and standardised as the IEEE 488 bus system.

Table 3.1 The ISO seven-layer model

Layer	*Description*	*Standards*
Physical	Defines the electrical and mechanical interfacing to a physical medium. Sets up, maintains and disconnects physical links. Includes hardware (I/O ports, modems, communication lines, etc.) and software (device drivers)	RS232-C RS442/443/449 V.24/V.28 V.10/V.11 X.21, X.21 bis, X.26, X.27, X.25 level 1
Data link	Establishes error-free paths over physical channel, frames messages, error detection and correction. Manages access to and use of channels. Ensures proper sequence of transmitted data	ANSI-ADCCP ISO-HDLC LAP DEC DDCMP IBM SDLC, BISYNC X.25 level 2
Network	Addresses and routes messages. Sets up communication paths. Flow control	USA DOD-IP X25, X75 (e.g. Tymnet, Telenet, Transpace, ARPANET, PSS)
Transport	Provides end-to-end control of a communication session. Allows processes to exchange data reliably	USA DOD-TCP IBM SNA DEC DNA
Session	Establishes and controls node-system-dependent aspects. Interfaces transport level to logical functions in node operating system	
Presentation	Allows encoded data transmitted via communications path to be presented in suitable formats for user manipulation	FTP JTMP FAM
Application (user)	Allows a user service to be supported, e.g. resource sharing, file transfers, remote file access, DBM, etc.	

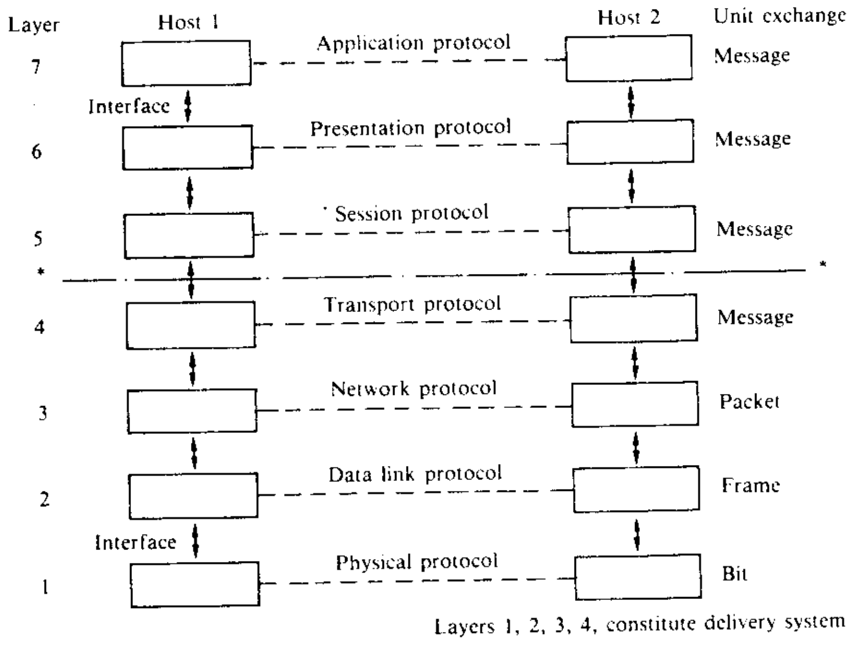

Figure 3.25 ISO seven-layer model.

The bus can connect up to a maximum of 15 devices and is only suited to laboratory or small, simple control applications.

The ISO (International Organisation for Standardisation) have promulgated a standard protocol system in the Open Systems Interconnection (OSI) model. This is a layered (hierarchical) model with seven layers running from the basic physical connection to the highest application protocol. The general structure is illustrated in Figure 3.25. The layers can be described as shown in Table 3.1.

3.9 SUMMARY

This chapter has provided a brief overview of some of the basic hardware ideas that are relevant to using computers in embedded control applications. We have concentrated on the basic ideas and not on particular microprocessors. In order to design a control system involving embedded computers you will need to obtain detailed knowledge about the particular microprocessor, microcomputer or microcontroller that you are going to use. In particular you will need to understand in detail its interface and where appropriate the range of interface support chips available for the particular processor. If you are going to be able to choose an appropriate device you will need to understand the important characteristics of a wide range of devices.

The key to understanding and coping with the complexities of the available hardware is to think in layers or hierarchies. An example of this was given in section 3.2.4 when we described the bus structure: we divided the discussion into physical, electrical and functional characteristics. Another example is the approach adopted by the ISO in its OSI communication model; detail that is not required at a higher level is hidden in the lower levels. As you will see later we will apply this approach to software, and will hide unwanted detail in low-level software modules.

An important aspect of interfacing is the timing of the transfer of data and the synchronisation of transfers. Timing diagrams of the form shown in Figure 3.4 are important and you need to be able to read and understand such diagrams.

EXERCISES

3.1 Why is memory protection important in real-time systems?
What methods can be used to provide memory protection?

3.2 A large valve controlling the flow of steam is operated by a dc motor. The motor controller has two inputs:

1. on/off control, 0 V = off, 5 V = on; and
2. direction, 0 V = clockwise, 5 V = anti-clockwise;

and two outputs:

1. fully open = 5 V;
2. fully closed = 5 V.

Show how this valve could be interfaced to a computer controlling the process.

3.3 A turbine flow meter generates pulses proportional to the flow rate of a liquid. What methods can be used to interface the device to a computer?

3.4 There are a number of different types of analog-to-digital converters. List them and discuss typical applications for each type (see, for example, Woolvet (1977) or Barney (1985)).

3.5 The clock on a computer system generates an interrupt every 20 ms. Draw a flowchart for the interrupt service routine. The routine has to keep a 24 hour clock in hours, minutes and seconds.

3.6 Twenty analog signals from a plant have to be processed (sampled and digitised) every 1 s. The analog-to-digital converter and multiplexer which is available can operate in two modes: automatic scan and computer-controlled scan. In the automatic scan mode, on receipt of a 'start' signal the converter cycles through each channel in turn.

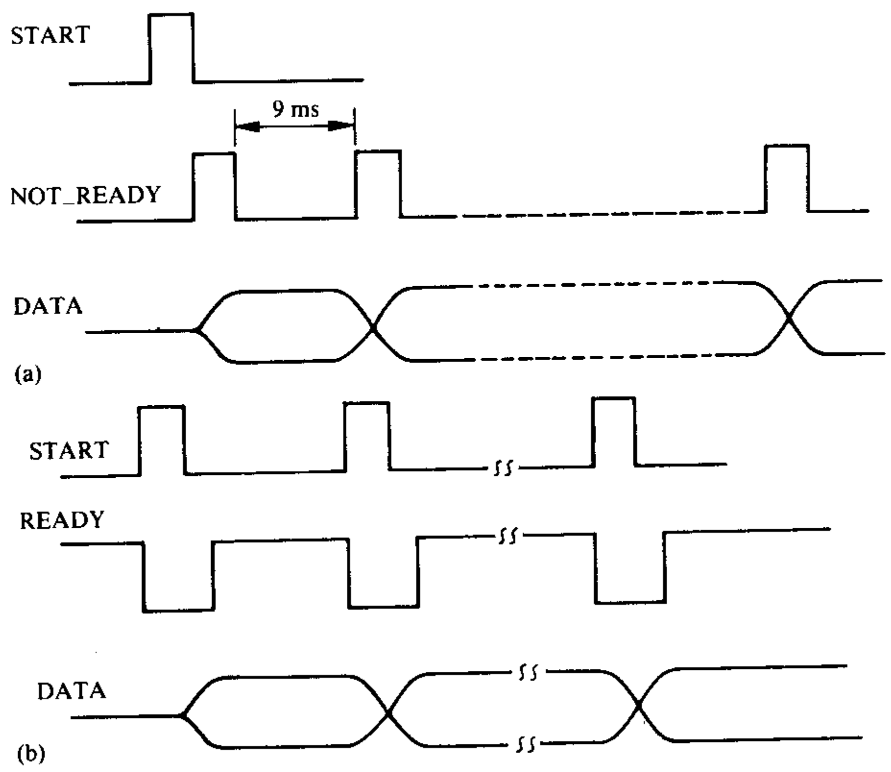

Figure 3.26 Timing diagram for Exercise 3.6.

The data corresponding to the channel sampled is available for 0.9 ms. The signal 'not-ready' is asserted during the conversion period and this indicates that the data is changing and should not be read by the computer. The timing is shown in Figure 3.26. In Mode 2 under computer-controlled scanning, the converter holds the data for each channel sampled until it receives a command from the computer to start the sampling of the next channel. To speed up the operation the multiplexer is switched to the next channel once the current channel has been sampled and before the computer reads the data for the current channel. The converter can be reset to start from Channel 1 by asserting a signal reset. The timing of this mode of operation is shown in Figure 3.26b. Consider the ways in which (a) polling and (b) interrupt methods can be used to interface the converter to a computer. Discuss in detail the advantages and disadvantages of each method.

3.7 We will assume that the simple heat process described in Chapter 1 has, in addition to a temperature sensor and heat controller, some logic signals and control switches. These are:

Plant controls:
heater on/off;

blower on/off; and
power on/off.
Plant signals:
overtemperature alarm; and
blower failure alarm.
The start-up sequence for the unit is:

1. Turn power on.
2. Turn blower on.
3. Wait 5 seconds.
4. Turn heater on.
5. DDC control action begins.

If at any time the overtemperature alarm becomes true, that is the signal level is set to logic 'high', the heater must be turned off but the blower kept running. If the blower failure alarm is detected, both the blower and the heater must be switched off. Draw a flowchart to show the sequence of operations to be carried out (a) for start-up and (b) in the event of failure.

3.8 The hot-air blower system described in Chapter 1 uses interrupts to indicate: power failure, printer ready, air temperature too high, VDU display ready, blower failure, clock signal and key pressed on the keyboard. Draw up a list of the priority order for the interrupts and explain the reasons for your choice of priority. Should any of the interrupts be connected to a non-maskable interrupt?

4

DDC Algorithms and Their Implementation

The main purpose of this chapter is to consider the methods used to implement simple digital control algorithms and some of the problems that arise in so doing. We shall take as an example a widely used and simple control algorithm: the PID or three-term control algorithm.

We shall consider:

- The digital form of the algorithm.
- The timing requirements.
- Integral wind-up and bumpless transfer.
- Choice of sampling rates.

The last two sections of the chapter introduce some more advanced control ideas concerned with finding the discrete equivalents of continuous controllers and the implementation of control algorithms designed using discrete control system design techniques. These sections can be omitted if you so wish.

4.1 INTRODUCTION

In Chapter 2 we introduced the idea of DDC (Direct Digital Control) and stated the differential equation for a PID controller:

$$m(t) = K_p[e(t) + 1/T_i \int_0^t e(t)dt + T_d de(t)/dt] \tag{4.1}$$

where $e(t) = r(t) - c(t)$, $r(t)$ is the desired value (set point), $c(t)$ the value of the variable being controlled and $m(t)$ the output from the controller. The differential equation is the time domain representation of the controller. The equivalent frequency domain representation is

$$G_c(s) = \frac{M(s)}{E(s)} = K_p\left(1 + \frac{1}{T_i s} + T_d s\right) \tag{4.2}$$

In the frequency domain the overall system of controller and plant can be represented by a block diagram as shown in Figure 4.1.

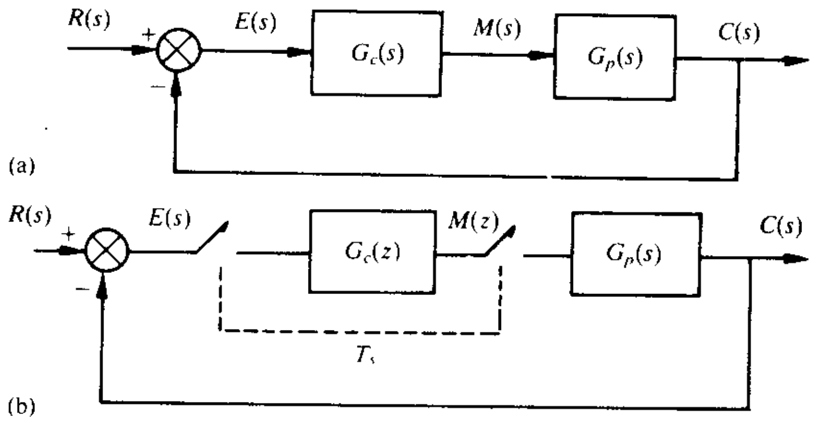

Figure 4.1 General form of a control system: (a) continuous form; (b) discrete form.

Both the time domain and frequency domain representations are continuous representations. To implement the controller using a digital algorithm we have to convert from a continuous to a discrete representation of the controller. There are several methods of doing this; the simplest is to use first-order finite differences. Considering the time domain version of the controller (equation 4.1) we replace the differential and integral terms by their discrete equivalents by using the relationships

$$\left.\frac{df}{dt}\right|_k = \frac{f_k - f_{k-1}}{\Delta t}, \quad \int e(t)dt = \sum_{k=1}^{n} e_k \Delta t \tag{4.3}$$

and hence equation 4.1 becomes

$$m(n) = K_p \left[T_d\left(\frac{e(n) - e(n-1)}{\Delta t}\right) + e(n) + \frac{1}{T_i}\sum_{k=1}^{n} e_k \Delta t \right] \tag{4.4}$$

where $m(n)$ represents the value of m at some time interval $n\Delta t$ where n is an integer.

By introducing new parameters as follows:

$$K_i = K_p(T_s/T_i)$$
$$K_d = K_p(T_d/T_s)$$

where $T_s = \Delta t =$ the sampling interval, equation 4.4 can be expressed as an algorithm of the form

$$\begin{aligned} s(n) &= s(n-1) + e(n) \\ m(n) &= K_p e(n) + K_i s(n) + K_d[e(n) - e(n-1)] \end{aligned} \tag{4.5}$$

where $s(n) =$ the sum of the errors taken over the interval 0 to nT_s.

4.2 IMPLEMENTATION OF THE BASIC PID ALGORITHM

Writing the code to implement the algorithm given in equation 4.5 is a simple job. The basic code statements are:

```
sn :=sn+en;
mn :=Kp*en+Ki*sn+Kd*(en-enOld);
enOld :=en;
```

These statements can be incorporated into a procedure such as:

```
PROCEDURE PIDControl(en: REAL; VAR mn: REAL);
BEGIN
   sn :=sn+en;
   mn :=Kp*en+Ki*sn+Kd*(en-enOld);
   enOld :=en;
END PIDControl;
```

If we assume that the plant output is obtained by using an ADC to sample and convert the output signal, that the actuator control signal is output through a DAC to the actuator and that procedures `ADC` and `DAC` are available to read the ADC and to send values to the DAC, then we can write a PID control module as shown in Example 4.1.

EXAMPLE 4.1

```
MODULE PIDcontroller;
(*
Title : PIDcontroller
File : PIDCONTR
Last Edit : 17 Jan 1992
Author : S.Bennett
*)
(*
This is the ideal controller. It ignores all
practical problems and all timing problems.
*)
FROM IOmodule IMPORT ADC, DAC;
FROM IO IMPORT KeyPressed;
CONST
   KpValue=1.0;
   KiValue=0.8;
   KdValue=0.3;
VAR
   sn,Kp,Ki,Kd,en,enOld,mn : REAL;
```

```
PROCEDURE PIDControl(en: REAL; VAR mn: REAL);
BEGIN
   sn := sn+en;
   mn := Kp*en+Ki*sn+Kd*(en-enOld);
   enOld := en;
END PIDControl;

BEGIN (* Main program *)
   sn:=0.0; (* initialise integral action term *)
   Kp:=KpValue;
   Ki:=KiValue;
   Kd:=KdValue;
   enOld:=ADC();
   REPEAT
      en:=ADC();
      PIDControl(en, mn);
      DAC(mn);
   UNTIL KeyPressed();
END PIDcontroller.
```

The above program ignores several important practical problems, for example it does not take into account the need to synchronise the calculation of `PIDControl` with 'real' time. As written the program makes the sampling interval T_s dependent on the speed of operation of the computer on which the program is run. For correct operation some means of fixing the sampling interval is required since the coefficients K_i and K_d are both calculated assuming a specific value of sampling interval. It is of course possible to change the algorithm to include the sampling interval T_s as a variable.

Other omissions include:

- bumpless transfer – that is, smooth transfer from manual to automatic control;
- actuator limiting and other forms of saturation – these lead to integral wind-up; and
- measurement and process noise.

The program also has the controller parameters built in as program constants; hence modification of the controller settings requires recompilation of the program.

In the next sections we examine some ways of modifying the program to deal with these problems and look at improved forms of the basic algorithm.

4.3 SYNCHRONISATION OF THE CONTROL LOOP

A typical feature of real-time programs is that once they have been started they run continuously until some external event occurs to stop them. We will emphasise this by using the infinite loop programming construct LOOP...END with an EXIT statement to indicate the terminating condition. The general form of a control program will be:

```
MODULE RealTimeControl;
(* declarations *)
BEGIN
   (* initialisation *)
   LOOP
      (* synchronisation *)
      (* get plant data *)
      (* control calculation *)
      (* EXIT condition check *)
      (* put control data to plant *)
   END (* loop *);
END RealTimeControl.
```

Synchronisation can be achieved by several different means such as:

- polling;
- external interrupt signals;
- ballast coding; and
- real-time clock signals.

Two methods, polling and external interrupt signals, were discussed in Chapter 3 (section 3.6). They rely on the plant (or some other unit external to the computer) sending a signal to indicate that it is time for a control action to take place. This signal must be sent at the sampling interval T_s chosen when starting the controller since, as can be seen from equation 4.4, the algorithm is correct only for a particular sampling rate. The difference between the two methods is that in polling the control computer repeatedly reads a value – normally a logical signal – whereas with an external interrupt the computer can be performing other computations; the action of the interrupt is to tell the computer to suspend whatever it is doing and carry out the control task.

4.3.1 Polling

We can write a synchronisation procedure which uses polling as follows:

```
PROCEDURE Synchronisation;
(* Use of polling for synchronisation *)
BEGIN
   LOOP
      WHILE NOT (Digin(SampleTime)) DO
         (* wait until time *)
      END (* while *);
   END (* loop *);
END Synchronisation;
```

In the above example it is assumed that a Boolean function, `Digin(Line)`, is available which reads the appropriate logical signal, in this case `SampleTime`, from the plant interface: When the procedure `Synchronisation` is called the computer waits in the `WHILE...DO` loop until the `Digin(SampleTime)` function returns a value true – this wait is referred to as a *busy wait* since no other computation can be carried out while the computer is waiting.

The polling method is simple to program and easy to design and use; however, because of the busy wait its use is restricted to small dedicated systems. An alternative method suitable for simple, dedicated, control systems is to use ballast coding (Hine and Burbridge, 1979).

4.3.2 Ballast Code

The idea of ballast coding is to make the loop time completely dependent on the internal operations of the computer and independent of external timing or synchronisation signals. The method involves finding the time taken to execute each possible path in the control loop of the program and adding code statements – ballast code – to make the execution time for each path equal. If necessary a further block of ballast code is added at the end to make the total execution time for the control loop equal to some desired execution time.

The method can be illustrated by considering the program structure shown in Figure 4.2. For each path (for example, A, $A1$, $A1.1$) the computational time for that particular path is calculated (or measured) and ballast code is added to each so as to make the computational time for each path equal. For path A, $A1$, $A1.1$ ballast A1.1 is added. Further ballast code can be added to make the total computational time equal to the sample interval; this is shown as Ballast B.

The method minimises the amount of external hardware required and is thus cost effective for systems that are to be produced in large quantities. An obvious problem is that any change in the code results in the need to adjust the ballast code segments. Also the technique cannot be used if interrupts are being used (why not?) and the code will have to be modified if the CPU clock rate is changed. As is the case with polling, the use of the ballast code technique prevents the computer system being used to carry out any other calculations while it is waiting to carry out the next control calculation.

4.3.3 External Interrupt

For small systems with a limited number of DDC loops (or other actions that require synchronisation), use of an external interrupt for synchronisation can be very effective. The control loop is written as an interrupt which is associated with a particular interrupt line. The interrupt line is activated by some external device – typically a clock. While the control loop is waiting to be activated other programs

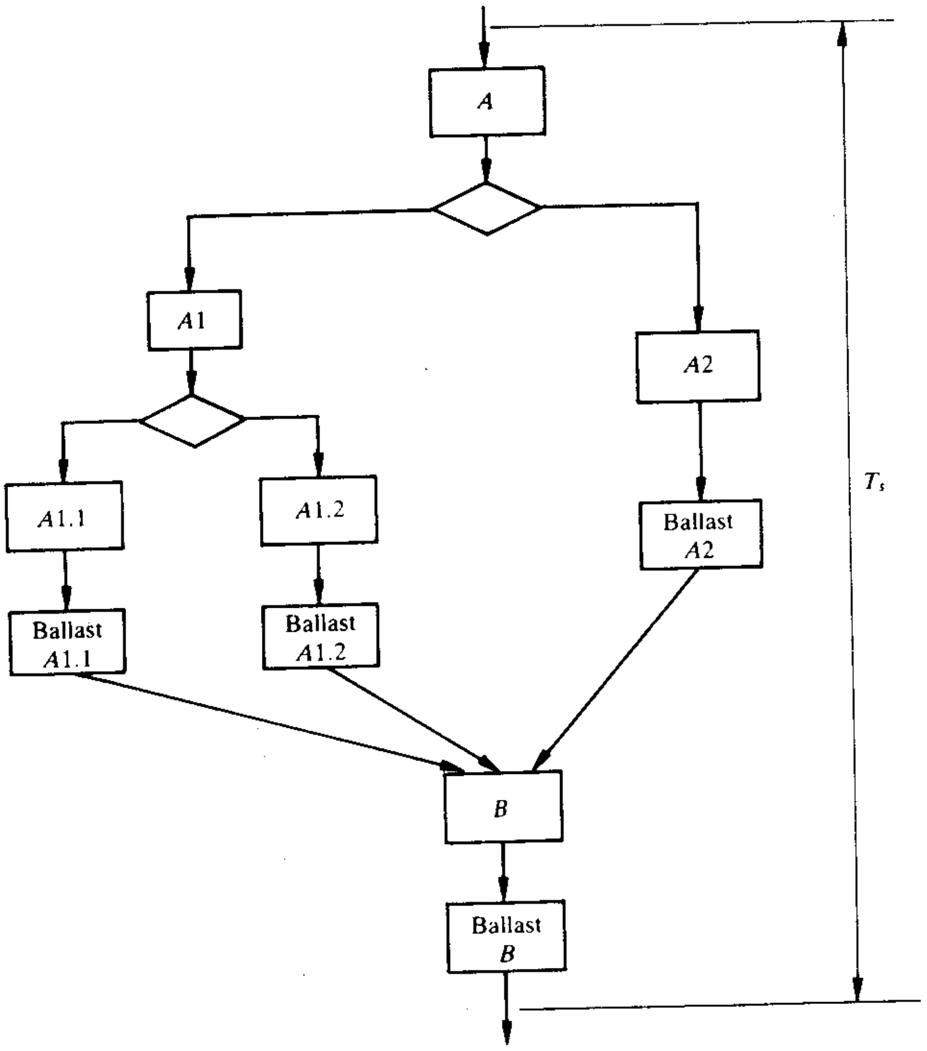

Figure 4.2 Ballast coding.

can be running. This form of operation is typically referred to as a foreground–background operation and is described in more detail later in this chapter.

4.3.4 Real-time Clock

The most general solution to the problem of timing a control loop is provided by adding a real-time clock to the computer system. Provision of a real-time clock involves the addition of some hardware components and some software. The IBM PC (and compatibles) as part of the BIOS (Basic Input Output System) provides a clock. The Module Timer, Example 4.2, shows how, using Modula-2, the clock values can be accessed. The time is returned in terms of the number of *ticks* of the clock, a tick being the resolution of the clock, that is the smallest interval of time the clock can measure.

EXAMPLE 4.2

```
DEFINITION MODULE Timer;
(*
Returns the value of the 'TICK'·clock.
*)
PROCEDURE Ticks():LONGCARD;
END Timer.
IMPLEMENTATION MODULE Timer;
(*
Returns the value of the 'TICK' clock.
*)
VAR
   TimerLow  [0:046CH] : CARDINAL;
   TimerHigh [0:046EH] : CARDINAL;
   Time      [0:046CH] : LONGCARD;
PROCEDURE LowTicks():CARDINAL;
(*
Reads TIMER_LOW from the ROM BIOS clock.
(Incremented every 1/18.2 seconds.)
*)
BEGIN
   RETURN TimerLow
END LowTicks;
PROCEDURE HighTicks():CARDINAL;
(*
Reads TIMER_HIGH from the ROM BIOS clock.
(Incremented every hour.)
*)
BEGIN
   RETURN TimerHigh
END HighTicks;
PROCEDURE Ticks():LONGCARD;
(*
Reads the complete ROM BIOS clock. (18.2 ticks/second.)
*)
BEGIN
   RETURN Time
END Ticks;
END Timer.
```

The code fragment, Example 4.3, illustrates how the timer module can be used to synchronise the control loop calculations to real time.

EXAMPLE 4.3

```
FROM Timer IMPORT Ticks;
CONST
   sT=20; (* time between samples in 'TICKS' *)
VAR
   time, NextSampleTime : LONGCARD;
BEGIN  (* Main program *)
   NextSampleTime := Ticks()+sT;
   time:=Ticks()+sT;
   LOOP
      WHILE Ticks() < NextSampleTime DO
         (* nothing *)
      END; (* of WHILE *)
      time:=Ticks();
      (* get plant input *)
      (* control calculation *)
      (* put plant output *)
      NextSampleTime := time+sT;
      IF KeyPressed()THEN EXIT;
      END (* IF *);
   END; (* of LOOP *)
```

This example uses a busy wait in the control loop – the `WHILE...DO` statement – and during this wait the clock is read continually and checked against the time for the next sample. Immediately on exiting from the wait the current value of time is saved in the variable `time`. At the end of the control calculation the time for the next sample is updated by adding the sample interval to the variable `time`. The interval between successive runs of the control loop is thus independent of the time taken in the control calculation. It is good practice to check that at the end of the control loop the value of `NextSampleTime` is later than the current time and to provide an error indication if this is not the case. A suitable check would be to add the code

```
IF NextSampleTime < Ticks() THEN RaiseError(timing)
END (* IF *);
```

Two examples of PID controllers are given below. You should study both of them carefully and how the method of synchronisation for each differs.

EXAMPLE 4.4

```
MODULE PIDcon2A;
(*
Title : PIDcontroller Example
File : PIDcon2A
*)
(*
This example illustrates a method for synchronising the PID
control calculation but ignores other practical problems.
*)
FROM IOmodule IMPORT ADC, DAC;
FROM Timer IMPORT Ticks;
FROM IO IMPORT KeyPressed;
FROM Error IMPORT RaiseError;
CONST
    KpValue=1.0;
    KiValue=0.8;
    KdValue=0.3;
    sT=20; (* time between samples in 'TICKS' *)
VAR
    sn,Kp,Ki,Kd,en,enOld,mn : REAL;
    time, NextSampleTime : LONGCARD;

BEGIN (* Main program *)
    sn:=0.0; (* initialise integrate action term *)
    Kp:=KpValue;
    Ki:=KiValue;
    Kd:=KdValue;
    en:=ADC();
    NextSampleTime := Ticks()+sT;
    time:=Ticks()+sT;
    LOOP
       WHILE Ticks() < NextSampleTime DO
          (* nothing *)
       END; (* of WHILE *)
       time:=Ticks();
       enOld:=en;
       en:=ADC();
       sn:=sn+en;
       mn:=Kp*en+Ki*sn+Kd*(en-enOld);
       DAC(mn);
       NextSampleTime := time+sT;
       IF NextSampleTime < Ticks() THEN
          RaiseError(timing);
       END (* IF *);
       IF KeyPressed()THEN EXIT;
       END (* IF *);
    END; (* of LOOP *)
 END PIDcon2A.
```

EXAMPLE 4.5

```
MODULE PIDcon2B;
(*
Title : PIDcontroller
File : PIDCON2B
*)
(*
This is an alternative way of providing
synchronisation to that given in MODULE PIDcon2A.
*)
FROM IOmodule IMPORT ADC, DAC;
FROM Timer IMPORT Ticks;
FROM IO IMPORT KeyPressed;
FROM Error IMPORT RaiseError;
CONST
   KpValue=1.0;
   KiValue=0.8;
   KdValue=0.3;
   sT=20; (* time between samples in 'TICKS' *)
VAR
   sn,Kp,Ki,Kd,en,enOld,mn : REAL;
   time:LONGCARD;
```

```
BEGIN (* Main program *)
   sn:=0.0; (* initialise integrate action term *)
   Kp:=KpValue;
   Ki:=KiValue;
   Kd:=KdValue;
   en:=ADC();
   time:=Ticks()+sT;
   LOOP
      WHILE Ticks() < time DO
         (* nothing *)
      END; (* of WHILE *)
      time:=time+sT;
      enOld:=en;
      en:=ADC();
      sn:=sn+en;
      mn:=Kp*en+Ki*sn+Kd*(en-enOld);
      DAC(mn);
      IF Time < Ticks() THEN
         RaiseError(timing);
      END (* IF *);
      IF KeyPressed()THEN EXIT;
      END (* IF *);
   END; (* of LOOP *)
END PIDcon2B.
```

4.4 BUMPLESS TRANSFER

Equation 4.5 implies that in the steady state, with zero error, the controlled variable $m(n)$ is equal to the value of the integral term $K_i s(n)$. Ideally in the steady state with zero error we would like the integral term to be zero, which would mean that $m(n)$ is also zero. In many applications the steady-state operating conditions require that $m(n)$ has some value other than zero. For example, a steam boiler may require the fuel line valves to be half open. In the case of the hot-air blower described in Chapter 1 a non-zero voltage has to be applied to the heater input for the heater to provide heat output. Therefore the normal practice is to modify equation 4.5 by adding a constant term (M) representing the value of the manipulated variable at the steady-state operating point, giving

$$m(n) = K_p e(n) + K_i s(n) + K_d[e(n) - e(n-1)] + M \qquad (4.6)$$

The quantity M can be thought of as setting the operating point for the controller. If it is omitted and integral action is present, the integral action term will compensate for its omission but there will be difficulties in changing smoothly, without disturbance to the plant, from manual to automatic control. There will also be the danger that on change-over, a large change (for example, in a valve position) will be demanded. Plant operating requirements usually demand that manual/automatic change-over be made in the so-called 'bumpless' manner. Bumpless transfer can be achieved by several means.

4.4.1 Method 1 – Preset Change-over Value

The value of M is calculated for a given steady-state operating point and is inserted either as a constant in the program or by the operator prior to the change-over from manual to automatic mode. The transfer to automatic mode is made when the value of the error is zero; at the time of change-over the integral term is set to zero and the output $m(n)$ equal to M. The problem with this technique is obvious: the predetermined value of M is correct only for one specified load. If the load is varying it may not be possible or convenient to make the change-over at the predetermined load value. If the error is not zero on change-over there will be a sudden change in the value of the manipulated variable due to the proportional action.

4.4.2 Method 2 – Tracking of Operator Setting

During operation under operator control the manipulated variable (m) is set from the operator's control panel and the computer system keeps track of the value. This may be done either by obtaining an analog or digital readout from the operator's control panel, or by reading the value of the input to the control actuator on the plant. In both cases it may be necessary to convert and scale the reading obtained to conform to the units being used for m inside the computer. At the point at which change-over is made, the value of m is stored in a variable mc. Two methods of transfer can be used:

1. M is not preset and change-over is made when the error (e) is zero; then $M = mc$. Or
2. M is preset to a value appropriate for the nominal level and change-over is made when the error is not zero. The integral action term needs to be set to an initial value

   $$s = mc - K_p ec - M$$

   where ec = error value at change-over.

4.4.3 Method 3 – Velocity Algorithm

The PID algorithm given by equation 4.5 is often referred to as the positional algorithm because it is used to calculate the absolute value of the actuator position.

An alternative form of the PID algorithm, the so-called velocity algorithm, is widely used to provide automatic bumpless transfer. The velocity algorithm gives the change in the value of the manipulated variable at each sample time rather than the absolute value of the variable. In continuous terms it can be obtained by differentiating equation 4.1, with respect to time, to give

$$\frac{dm(t)}{dt} = K_p\left(\frac{de(t)}{dt} + \frac{1}{T_i}e(t) + T_d\frac{d^2e(t)}{dt^2}\right) \tag{4.7}$$

The difference equation can be obtained either by applying backward differences to equation 4.7 or by finding $m(n) - m(n-1)$ using equation 4.4 which gives

$$\begin{aligned}\Delta m &= m(n) - m(n-1)\\ &= K_p\left([e(n) - e(n-1)] + \frac{\Delta t}{T_i}e(n) + \frac{T_d}{\Delta t}[e(n) - 2e(n-1) + e(n-2)]\right)\end{aligned} \tag{4.8}$$

Rearranging equation 4.8 gives

$$\Delta m(n) = K_p\left[\left(1 + \frac{T_s}{T_i} + \frac{T_d}{T_s}\right)e(n) - \left(1 + 2\frac{T_d}{T_s}\right)e(n-1) + \frac{T_d}{T_s}e(n-2)\right] \tag{4.9}$$

Writing

$$\begin{aligned}K_1 &= K_p(1 + T_s/T_i + T_d/T_s)\\ K_2 &= -(1 + 2T_d/T_s)\\ K_3 &= T_d/T_s\end{aligned}$$

equation 4.9 becomes

$$\Delta m(n) = K_1e(n) + K_2e(n-1) + K_3e(n-2) \tag{4.10}$$

which is easily programmed.

Because it outputs only a change in the controller position this algorithm automatically provides bumpless transfer. However, if a large standing error exists on change-over the response of the controller may be slow, particularly if the integral action time is long, that is with a large value of T_i.

4.4.4 Comparison of Position and Velocity Algorithms

Comparing the position algorithm (equation 4.5) and the velocity algorithm (equation 4.10) shows that the latter is simpler to program and is inherently safer in that large changes in demanded actuator position are unlikely to occur. Frequently the maximum value which $m(n)$ can take is limited, thus ensuring that sudden large changes, for example in valve position or motor speed, are avoided. These sudden changes can occur if the measured signal is noisy or if the set point is changed. A method of dealing with noisy measurements is to use a fourth-order difference algorithm to approximate de/dt and this is explained in the next section. The disturbance caused by set point changes can be reduced by modifying the algorithm to use the set point r and the measured output c rather than the error signal e.

In the standard algorithm, based on the use of error e, the value of the set point appears in the derivative term and any change in value is differentiated; hence a sudden step change can cause a large disturbance. If in equation 4.9 we let $e(n) = r - c(n)$ (note that r, the set point, is assumed to be constant), then the equation becomes

$$\Delta m(n) = K_p\Big([c(n-1) - c(n)] + \frac{T_s}{T_i}(r - c(n)) + \frac{T_d}{T_s}[2c(n-1) - c(n-2) - c(n)]\Big) \tag{4.11}$$

Changes in the set point are then accommodated by simply changing the value of the constant r.

The set point r appears only in the integral term and hence the controller must always include integral action. For security of operation a check must be included in the program to prevent the T_s/T_i parameter being set to zero or some very small value.

Disturbances can also be caused by on-line parameter changes since with a digital algorithm a large parameter change can be introduced in a single step. Consider the basic form of the PID algorithm

$$\begin{aligned} s(n) &= s(n-1) + e(n) \\ m(n) &= K_p e(n) + K_i s(n) + K_d[e(n) - e(n-1)] \end{aligned} \tag{4.12}$$

If we make a change to the integral action time T_i clearly, unless $s(n)$ is zero, there will be a step change in output since $K_i = K_c(T_s/T_i)$. We can avoid this either by limiting the rate at which a parameter changes or by altering the algorithm as shown below:

$$\begin{aligned} s(n) &= s(n-1) + e(n)(T_s/T_i) \\ m(n) &= K_p e(n) + K_p s(n) + K_d[e(n) - e(n-1)] \end{aligned} \tag{4.13}$$

By writing the algorithm in this form the effects of making a change to T_i are much reduced. Changes to K_p and K_d cause much smaller disturbances unless the error is large and/or the rate of change of error is large.

4.5 SATURATION AND INTEGRAL ACTION WIND-UP

In practical applications the value of the manipulated variable $m(n)$ is limited by physical constraints. A valve cannot be more than fully opened, or more than fully

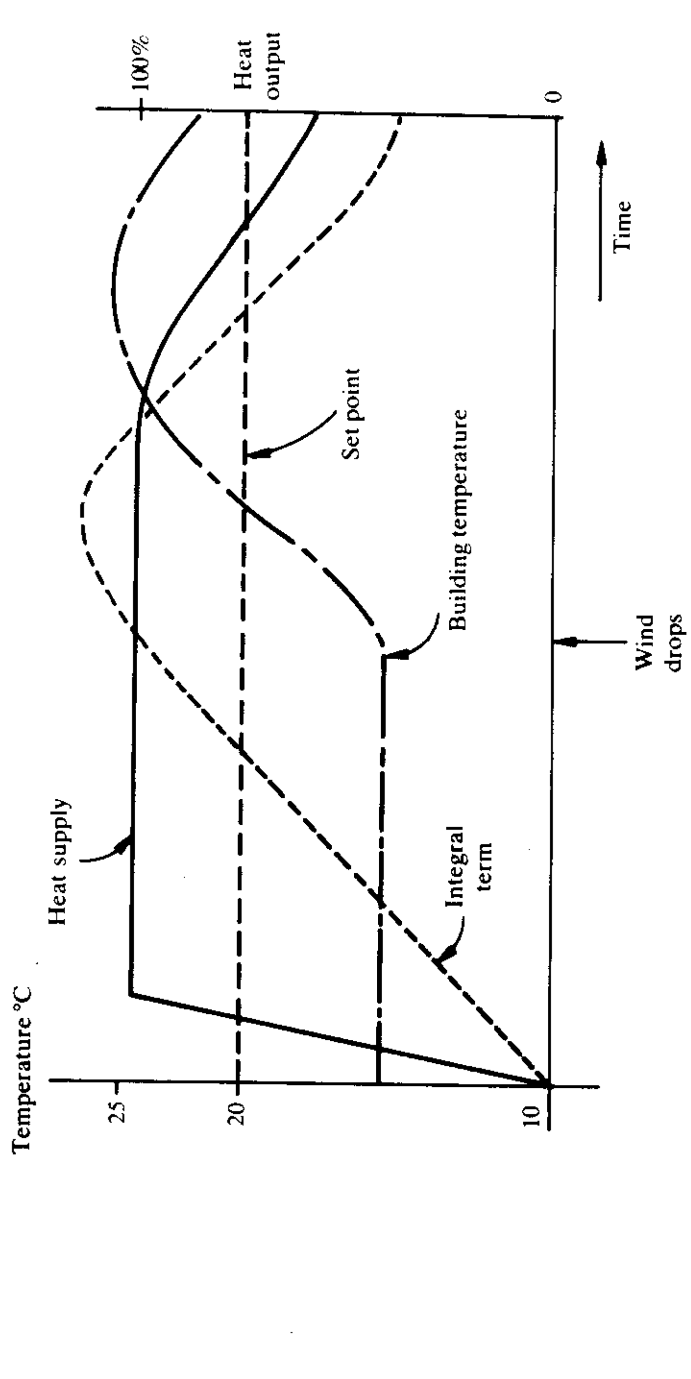

Figure 4.3 Illustration of integral wind-up action.

closed; a thyristor-controlled electric heater can supply only a given maximum amount of heat and cannot supply negative heat. If the value of the manipulated variable exceeds the maximum output of the control actuator effective feedback control is lost: good plant design should ensure that this only occurs in unusual conditions.

A simple example of what can happen is provided by considering a building heating system. The capacity of such a system is usually chosen to cope with an average winter: if extreme low temperature and high winds coincide, the system, even when operating at maximum capacity, will not be able to maintain the desired temperature. Under these conditions a large standing error in temperature will exist. If a PI controller is used, then because there is a standing error, the integral term will continue to grow; that is, the value of $s(n)$ in equation 4.5 will be increased at each sample time. Consequently the value of the manipulated variable will increase and the demanded heat output will continually increase, but since this will already be a maximum, the demand cannot be met. The changes are shown in Figure 4.3. If the wind drops and the outside temperature increases, then the building temperature will increase and eventually reach the desired temperature. The value of $s(n)$, the integral term, will, however, still be large, since it will not be reduced until the building temperature exceeds the demanded temperature. As a consequence the integral term will continue to keep the demanded heat output at its maximum value even though the building temperature is now higher than desired.

The effect is called *integral wind-up* or *integral saturation* and results in the controller having a poor response when it comes out of a constrained condition. Many techniques have been developed for dealing with the problems of integral wind-up and the main ones are:

- fixed limits on integral term;
- stop summation on saturation;
- integral subtraction;
- use of velocity algorithm; and
- analytical method.

4.5.1 Fixed Limits

A maximum and minimum value for the integral summation is fixed and if the term exceeds this value it is reset to the maximum or minimum as appropriate. The value often chosen is the maximum/minimum value of the manipulated variable; thus if $s_{\max} = m_{\max}$ and $s_{\min} = m_{\min}$ then the coding in the `PROCEDURE PIDControl` in Example 4.1 could be modified as shown in Example 4.6.

EXAMPLE 4.6

```
PROCEDURE PIDControl (en: REAL; VAR mn: REAL);
   sn :=sn+en; (* integral summation *)
   IF sn > smax THEN
      sn :=smax
   ELSE IF sn < smin THEN
           sn :=smin;
        END (* ELSE IF *)
   END (* IF *)
   mn :=Kp*en+Ki*sn+Kd*(en-enOld);
   enOld :=en;
END PIDControl;
```

4.5.2 Stop Summation

In this method the value of the integrator sum is frozen when the control actuator saturates and the integrator value remains constant while the actuator is in saturation. The scheme can be implemented either by freezing the summation term when the manipulated variable falls outside the range $m_{\min}$ to $m_{\max}$ or by the use of a digital input signal from the actuator which indicates that it is at a limit.

Both of the above methods stop the integral term building up to large values during saturation but both have the disadvantage that the value of the integral term, when the system emerges from saturation, does not relate to the dynamics of the plant under full power. Consequently the controller offset (provided by the integral term) lags behind the offsets required by the plant and load as the set point is reached.

The stop summation technique gives a better response if the integral term is unfrozen once the sign of the error changes. The sign of the error will change before the actuator comes out of saturation. Assume that a positive error drives the actuator towards its upper limit; then the behaviour required is:

Actuator	*Error*	*Integral summation*
upper limit	+	stopped
upper limit	−	active
normal	+	active
normal	−	active
lower limit	+	active
lower limit	−	stopped

The procedure becomes

```
PROCEDURE PIDControl (en: REAL; VAR mn: REAL);
    StopSummation:=((mn > mnmax) AND (en > 0.0))
        OR ((mn < mnmin) AND (en < 0.0));
    IF NOT (StopSummation) THEN sn:=sn+en;
    END (* IF *)
    mn:=Kp*en+Ki*sn+Kd*(en-enOld);
    enOld:=en;
END PIDControl;
```

4.5.3 Integral Subtraction

The idea behind this method is that the integral value is decreased by an amount proportional to the difference between the calculated value of the manipulated variable and the maximum value allowable. The integral summation expression

$$s(n) = s(n-1) + e(n)$$

is replaced by

$$s(n) = s(n-1) - K[m(n) - m_{\max}] + e(n)$$

The integral sum is thus decreased by the excess actuation and increased by the error. The rate of decrease is dependent on the choice of the parameter K; if it is not properly chosen then a continual saturation/desaturation oscillation can occur.

The method can be modified to stop the addition of the error part during saturation if a logic signal from the actuator indicating saturation or no saturation is available. In this case the value of the integral sum begins to decrease as soon as the actuator enters saturation and continues to decrease until it comes out of saturation, at which point integral summation begins again. The benefit of this method is that the system comes out of saturation as quickly as possible; there is, however, no attempt to match the integral term to the requirements of the plant and the value of K must be chosen by experience rather than by reference to the plant characteristics.

4.5.4 Velocity Algorithm

It is often stated that integral wind-up can be avoided by the use of the velocity algorithm since the integral action is obtained by a summation of the increments in the output device, either at the actuator or at a device connected to the actuator, and it is this device which is subject to limiting. There is therefore an automatic integral limit which prevents a build-up of error. However, as soon as the error changes sign the actuator will come off its limit and hence at desaturation the integral term is lost.

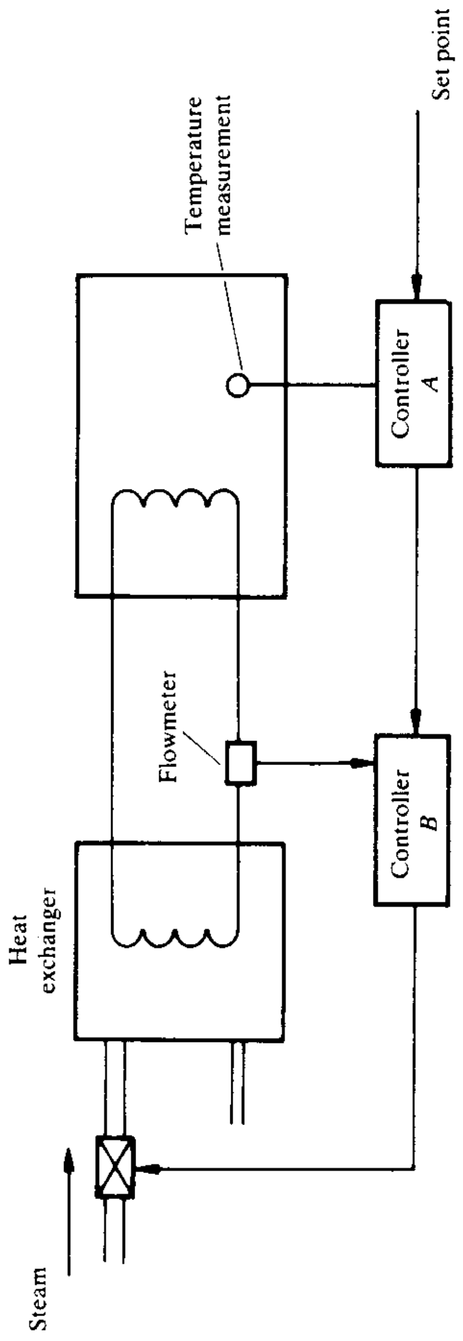

Figure 4.4 Cascade control system.

When controllers are cascaded it must not be assumed that the use of the velocity algorithm will prevent integral wind-up. In the system shown in Figure 4.4, controllers A and B are assumed to use the velocity algorithm and both are assumed to employ PI control. Controller A is used to adjust the steam flow to a heat exchanger in order to maintain a particular water temperature in the hot-water supply used to heat a room. The demanded water temperature is set by controller B. Suppose on a cold day controller B demands a water temperature of 60°C but the best the heat exchanger can do is to provide water at 55°C. The effect will be that the room temperature will remain too low and the integral sum will begin to grow and hence the set point of the controller will continually be increased until it reaches its maximum limit. If the steam valve is fully open there is no action controller A can take. If now the external temperature increases there will be a delay, during which the room temperature might rise well above the desired temperature, before the set point of controller A is reduced to correct the overshoot. In order to avoid this the master controller must know when the subsidiary loop is at a limit.

4.5.5 Analytical Approach

This method has been developed by Thomas, Sandoz and Thomson (1983) and it uses knowledge of the plant to set the integral sum term approximately to the correct value at the point of desaturation such that the normal linear response from steady state is achieved. This is shown in Figure 4.5. For a system of the form shown in Figure 4.6, that is a first-order plant with a PI controller, the integral sum at the time τ when the system desaturates is given by

$$I(\tau) = \frac{1}{K_pK_c}\,[c(\tau) + sK_pL(s)] \tag{4.14}$$

If it is assumed that the load $L(s)$ is constant, or slowly varying, then

$$L(s) \approx L_0/s \tag{4.15}$$

and hence equation 4.14 becomes

$$I(\tau) = \frac{1}{K_pK_c}\,[c(\tau) + K_pL_0] \tag{4.16}$$

When the control actuator desaturates the integral value should be set to the value which it would have been holding in the steady state at $c(\tau)$ and then the remaining step $e(\tau)$ will follow as in the linear case. The value of $c(\tau)$ is not known since the time, τ, of desaturation is not known; however, if prior to the control calculation, the integral term $I(\tau)$ is set using equation 4.16 above with the $c(\tau)$ for $t < \tau$, then at the instant of desaturation $I(t) = I(\tau)$. If the actuator is not in saturation then the normal integration of error takes place. This scheme is shown in the program segment in Example 4.7 where the `REAL` variables `Load` and `KPKC` are used for K_pL_0 and K_pK_c respectively.

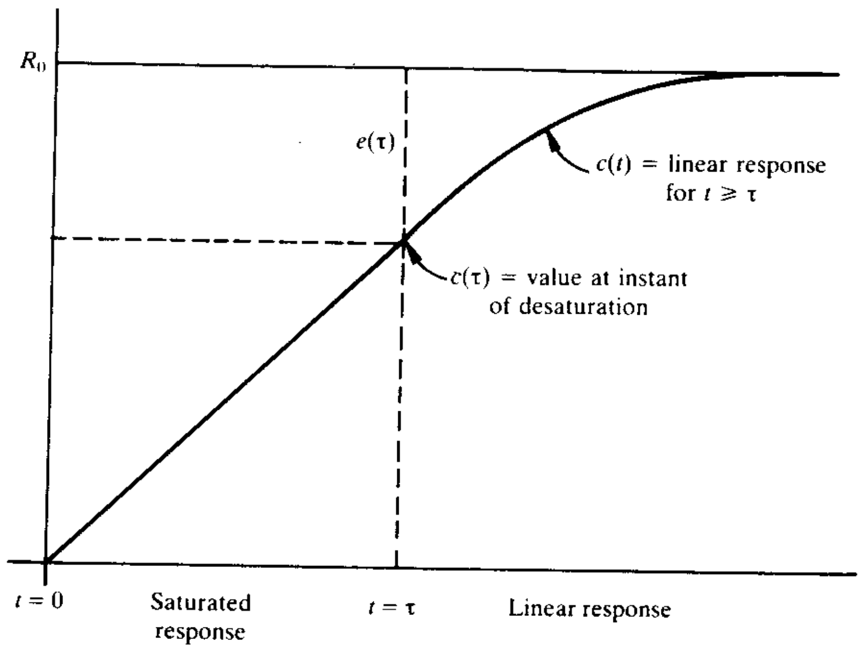

Figure 4.5 Saturated and linear regions of first-order responses.

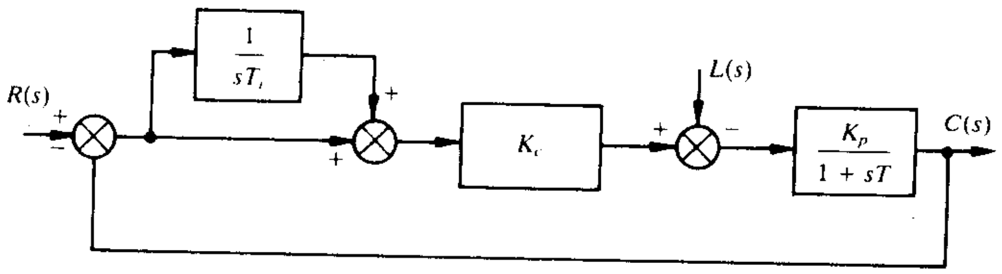

Figure 4.6 PI control of first-order plant.

EXAMPLE 4.7

```
LOOP
   cn :=ADC; (* ADC returns value of plant output
*)
   en :=cn-setpoint;
   IF mn>mnmax THEN
      sn :=(cn+load)/KPKC
   ELSE
      sn :=sn +en; (* integral summation *)
   END (* IF *)
   mn :=Kp*en+Ki*sn+Kd*(en-enOld);
   DAC(mn);
   enOld :=en;
END (* LOOP *)
```

4.6 TUNING

The first papers on methods of tuning analog three-term controllers were those of J.G. Ziegler and N.B. Nichols which were published in 1942 and 1943. Since then there has been extensive work on extending and developing tuning methods. In recent years direct tuning by plant engineers has largely been replaced by autotuning controllers. The basic tuning methods are covered in most texts on control engineering: for adaptation to digital implementations see for example Leigh (1992), Astrom and Wittenmark (1984), Takahashi *et al.* (1970). All of the methods are based on simple plant measurements and on the assumption that the plant can be modelled by the transfer function

$$G(s) = \frac{ke^{-Ls}}{(1 + sT_p)} \tag{4.17}$$

The value of the parameters in the above plant model can be obtained from a simple open-loop step response as shown in Figure 4.7.

For the analog system the tuning problem is, given R, L or T_r, to choose K_p, K_i, K_d so as to minimise (or maximise) some performance criteria (for example, the integral of absolute error, IAE).

The standard Ziegler–Nichols rules are:

	K_p	T_i	T_d
P	$1/RL$		
PI	$0.9/RL$	$3L$	
PID	$1.2/RL$	$2L$	$0.5L$

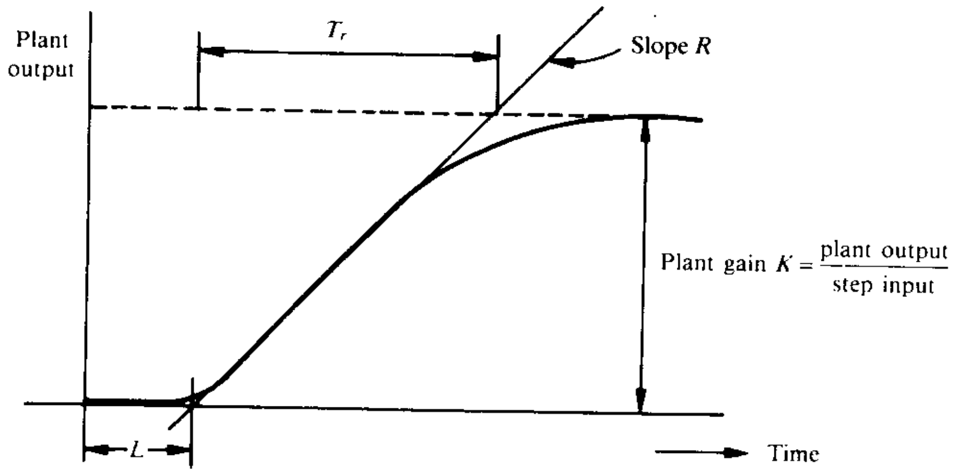
Plant output
T_r
Slope R
Plant gain $K = \frac{\text{plant output}}{\text{step input}}$
L
Time

Figure 4.7 Process reaction curve.

The alternative method is to close the feedback loop and use a proportional controller. The gain of the controller is turned up until the system just begins to oscillate with a steady amplitude. The tuning parameters are then calculated in terms of the gain K_m and the period of the oscillation T_p. The settings are

	K_p	T_i	T_d
P	$0.5\,K_m$		
PI	$0.45\,K_m$	$T_p/1.2$	
PID	$0.6\,K_m$	$T_p/2$	$T_p/8$

The digital implementation of the three-term controller involves additional variables: q the unit of quantisation and T_s the sampling time; hence the performance function becomes

$$J = f(q,\ T_s,\ K_p,\ K_i,\ K_d,\ L,\ T_p)$$

The size of the quantisation is not normally a problem in industrial process control since the control computation can be done using either real numbers or fixed point arithmetic. It becomes greater in, for example, aircraft controls and weapons systems where time constants are shorter and, in order to obtain the necessary computational speed, limited wordlength arithmetic has to be used. The problems are discussed extensively by Katz (1981) and Leigh (1992); J.B. Knowles has devised a design method which takes into account quantisation and sample rates (see Bennett and Linkens, 1984).

Smith (1972) has suggested that when using tuning tables based on L values

$(L + T_s/2)$ should be used rather than L, in order to take into account the delay caused by sampling. He also notes that as the value of $T_s/2$ approaches the dead time, L, the performance deteriorates.

If the PID algorithm is expressed in the form given in equation 4.10 but with

$$K_i = K_c T_s / T_i \quad \text{and} \quad K_d = K_c T_d / T_s$$

then Takahashi *et al.* (1970) give the following rule for tuning the parameters:

$$K_p = \frac{1.2}{R(L + T_s)} - \frac{1}{2K_i}$$

$$K_i = \frac{0.6T_s}{R(L + T_s/2)^2}$$

$$K_d = \frac{0.5}{RT_s} \quad \text{to} \quad \frac{0.6}{RT_s}$$

When $T_s = 0$ the above rule converges to the standard Ziegler–Nichols result

$$K_p = 1.2/RL, \quad 1/T_i = 0.5/L, \quad T_d = 0.5L$$

It should be noted that the quality of the control deteriorates when T_s increases relative to the dead time, L, and that the tuning rule fails when L/T_s is very small.

4.7 CHOICE OF SAMPLING INTERVAL

Intuitively we might assume that if we decrease the sampling interval then as T_s tends to zero the system will asymptotically converge towards the performance of the equivalent analog system. However, this is not the case since the digital computation has a finite resolution. Thus as the sample interval decreases the change between successive values becomes less than the resolution of the system and hence information is lost. The interaction between sampling interval and arithmetic resolution is complex and a detailed analysis can be found in Katz (1981) and in Williamson (1991); for a shorter summary of the effects see Leigh (1992).

Conversely increasing the sampling interval can result in destabilising the feedback loop, in loss of information (the sampling effect), and in a loss of accuracy of the control algorithm. Various empirical rules have been given for choosing a sample interval, some of which are listed below:

1. Dominant plant time constant: If the dominant plant time constant is T_p then choose T_s such that

   $$T_s < T_p/10$$

   This is a widely used rule but as Leigh points out it is dangerous under conditions where a high closed-loop performance is forced on a system with a low open-loop performance.

2. Assumption of Ziegler–Nichols plant model: If the plant model is as given in equation 4.17 then the following suggestions are given in the literature:
   (a) general $0.05 < T_s/L < 0.3$,
   (b) large dead time L set $T_i/T_s = 2$,
   (c) small dead time L set $T_i/T_s = 6$,
   (d) choose sampling interval such that $5 < T_d/T_s < 10$.
3. Closed-loop performance requirements: If the closed-loop system is required to have either a settling time of T_{ss} or a natural frequency of ω_n then choose T_s such that
   (a) $T_s < T_{ss}/10$,
   (b) $\omega_s > 10\omega_n$ where $\omega_s = 2\pi/T_s$.

4.8 PLANT INPUT AND OUTPUT

So far we have assumed that the input signal obtained from the plant and used to calculate the error signal e is a clean signal that is in the correct form required for the control algorithm. In practice this is not the case. The signal may be a noisy signal, it may require the application of a calibration factor and it may need scaling. Similarly we have assumed that the output from the control algorithm can be sent directly to the actuator and again this is not generally the case: the value may need scaling and converting to a different form of output, for example in process control applications many actuators require a pulse of varying duration to operate them rather than a signal of varying amplitude.

A way of dealing with the practical details of input and output from the plant is to divide the software into segments as shown in Figure 4.8. The plant input segment is visualised as providing a plant image that is a filtered, scaled and calibrated version of the plant input signal. The control algorithm uses this and calculates the actuator command signal which it places in the output image. The plant output module converts this into a form required to drive the real actuator on the plant.

4.8.1 Noise

In an analog control system, a small amount of high-frequency noise on the measured signal usually does not cause any problems since the dynamic components in the system act as low-pass filters and attenuate the noise. If sampling is involved, high-frequency noise may produce a low-frequency disturbance due to folding or aliasing (see, for example, Leigh, 1992; Kuo, 1980). The low-frequency disturbance has the same sample amplitude as the original noise and its frequency is the

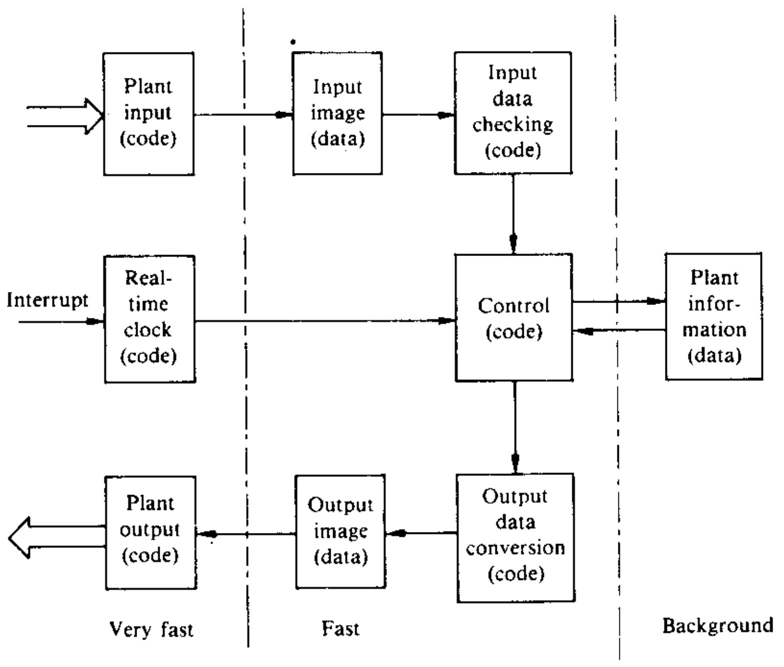

Figure 4.8 Basic division control software.

difference between the original noise frequency and a multiple of the sampling interval. To reduce this effect the measurement signal must be filtered using an analog filter before it is sampled.

For many industrial applications a simple first-order filter

$$G_f(s) = \frac{1}{1 + T_f s}$$

is satisfactory. The choice of $T_f = T_s/2$ (where T_s is the sampling time for the controller) will reduce the aliasing effect by about 90% for white noise. For example, if the bandwidth of the system being controlled is 0 to 2 Hz then from the Nyquist sampling theory we need to sample at a minimum frequency of 4 Hz if we are to be able to reconstruct the signal. In practice we would choose to sample at ten times the bandwidth and hence need to sample at 20 Hz which corresponds to a sampling interval T_s of 0.05 s. To remove high-frequency noise prior to sampling we need to use a filter with a time constant of $T_f = T_s/2$, that is 0.025 s.

Where T_f is small, analog filters can easily be constructed from passive elements. If T_f is greater than a few seconds then combined digital and analog filtering is used. The arrangement is illustrated in Figure 4.9 and the analog filter is used to remove the high-frequency noise and hence reduce the required sampling rate.

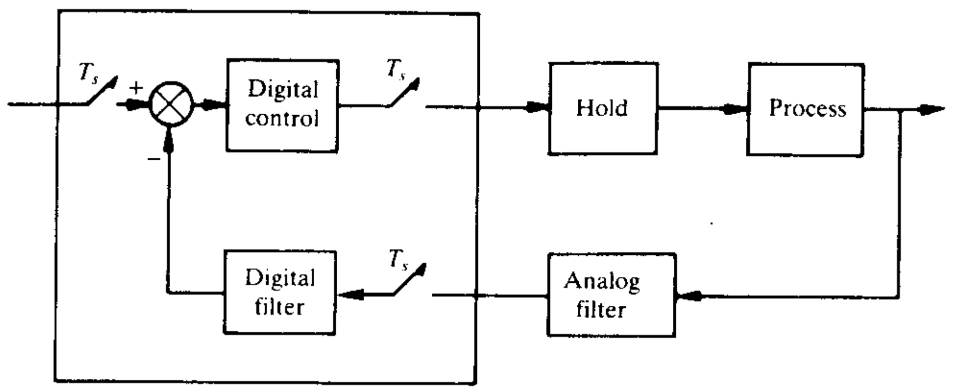

Figure 4.9 General control system with filtering.

For a first-order lag filter

$$T_f \frac{dx}{dt} + x = u$$

with input u, output x and filter sampling interval T_{fs} the numerical approximation is

$$x(n+1) = [1 - \exp(-T_{fs}/T_f)]x(n) + \exp(-T_{fs}/T_f)u(n)$$

Introducing $a = \exp(-T_{fs}/T_f)$ and applying a backwards time shift gives

$$x(n) = (1-a)x(n-1) + au(n-1)$$

It can be seen that if $a = 1$ there is no smoothing and if $a = 0$ the current measurement (input) is not used.

The sample interval T_{fs} for the input to the digital filter has to be made smaller than the time constant T_f of the filter and hence several samples of the measurement signal are taken for each output of the controller. The time constants of analog pre-filters are usually small and do not significantly degrade the overall performance. Excessively large values of T_f should be avoided.

4.8.2 Actuator Control

In designing and implementing real controllers the characteristics of the actuators used to control the process are important. The normal practice is to take the value of the manipulated variable ($m(n)$ or $\Delta m(n)$) as the input to an actuator control module as shown in Figure 4.10. In some instances there will be local feedback of the actuator position as shown by the dotted line in Figure 4.10. The advantage of this approach is that the basic controller and the detailed control of the actuator are separated. If the actuator is modified or changed then only the actuator controller module is affected.

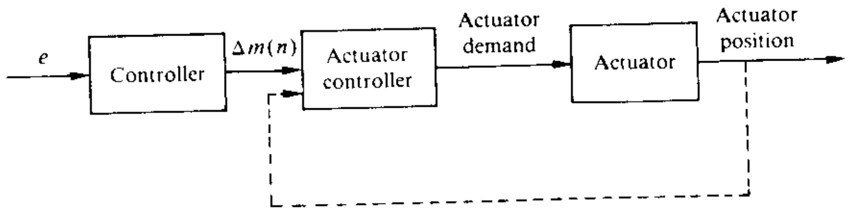

Figure 4.10 Actuator control.

4.8.3 Computational Delay

Analog-to-digital conversions, all computations in the computer – digital filtering, calculation of the control value – and digital-to-analog conversion all take a finite amount of time. There is thus a time difference – the computational delay – between sampling the plant output and changing the value of the actuator. The value of this delay depends on the computations carried out, on the processor and input/output speed of the computer and on the order in which certain operations are done. Intuitively most people will order the digital controller calculations as follows:

```
1 read plant input data
2 calculate control output
3 send output to actuator
```

If the kth sample of the plant input is measured at some time $t(k)$ then the kth sample of the actuator output is sent out at some time $t(k)+h$ where h is the computational delay time (note $h < T_s$). In other words a plant measurement $ct(k)$ gives rise to an output $u[t(k)+h]$. In general h will be variable as it will depend partially on the data values. If the calculations are ordered in this way then h is made as small as possible by performing as few operations as possible between getting the plant input and producing the output.

An alternative way of ordering the computations is as follows:

```
1 send to plant the control value for sample interval (k-1)
2 read plant input data for sample interval k
3 calculate control output for kth sample interval.
```

This approach produces a constant computational delay equal to the sample interval T_s. The advantages are that only statements 1 and 2 above need to be tightly synchronised to the real-time clock; statement 3, the control calculation, can be done at any time providing that the calculation is completed within the sample interval T_s and that the computational delay is constant. Whichever approach is used it is normal to take the computational delay into account by including the delay h or T_s as part of the plant model.

Astrom and Wittenmark (1984) suggest that when using the second method you should order the operations as

```
1 read plant input data for sample interval k
2 send to plant the control value for sample interval (k-1)
3 calculate control output for kth sample interval
```

so as to avoid the risk of electrical cross-coupling. If you do this you should split the processing of the input data so that only the actual read operations are carried out before you do the output to the actuator; any linearisation or other operations on the input data should be done after sending the previous control value to the actuator.

4.9 IMPROVED FORMS OF ALGORITHM FOR INTEGRAL AND DERIVATIVE CALCULATION

The positional and velocity algorithms considered so far use first- and second-order differences respectively to compute the derivative terms. Since differentiation or its numerical equivalent is a 'roughening' process – it accentuates noise and data errors – some form of smoothing, that is filtering, is required. Smoothing can be obtained by using a difference technique which averages the value over several samples. One such technique which has been used is the four-point central difference method (Bibbero, 1977; Takahashi *et al.*, 1970). This gives

$$\frac{de}{dt} = \frac{\Delta e}{T_s} = \frac{1}{6T_s}\,[e(n) + 3e(n-1) - 3e(n-2) - e(n-3)] \tag{4.18}$$

Substituting for $T_d/T_s[e(n) - e(n-1)]$ in equation 4.4 gives

$$m(n) = K_p\left(\frac{T_d}{6T_s}\,[e(n) + 3e(n-1) - 3e(n-2) - e(n-3)] + e(n) + \frac{T_s}{T_i}\sum_{k=1}^{n} e(k)\right) \tag{4.19}$$

The position algorithm using the above technique for the derivative term is thus

$$s(k) = s(k-1) + e(k)$$
$$m(k) = p_1 e(k) + p_2 e(k-1) - p_2 e(k-2) - p_3 e(k-3) + p_4 s(k) \tag{4.20}$$

where

$$p_1 = 1 + K_p T_d/6T_s$$
$$p_2 = 1 + K_p T_d/2T_s$$
$$p_3 = 1 + K_p T_d/6T_s$$
$$p_4 = 1 + K_p T_d/T_i$$

Improvements can also be made to the accuracy of the integration calculation by using the trapezoidal rule instead of the rectangular rule. If this is done, equation 4.4 can be written as

$$m(n) = K_p'\left[e(n) + \left(\frac{T_s}{T_i'}\sum_{k=1}^{n}\frac{e(k)+e(k-1)}{2}\right) + \frac{T_d'}{T_s}\,[e(n)-e(n-1)]\right] \quad (4.21)$$

and in velocity algorithm form

$$\Delta m(n) = K_p'\left[\left(1+\frac{T_s}{2T_i'}+\frac{T_d'}{T_s}\right)e(n) + \left(\frac{T_s}{2T_i'}-\frac{2T_d'}{T_s}-1\right)e(n-1) + \frac{T_d'}{T_s}\,e(n-2)\right] \quad (4.22)$$

If this is compared with the velocity algorithm using rectangular integration we find that

$$K_p = K_p'(1 - T_s/2T_i')$$
$$T_i = T_i' - T_s/2$$
$$T_d = 2T_d'T_i'/(2T_i' - T_s)$$

Hence if the appropriate values for the coefficients are used the form of the two algorithms is identical.

4.10 IMPLEMENTATION OF CONTROLLER DESIGNS BASED ON PLANT MODELS

In the early days of computer control one of the justifications for using digital control instead of analog control was the ease with which more complex control algorithms could be introduced. In particular control algorithms could be designed on the basis of accurate plant models instead of relying on the Ziegler–Nichols assumption that the plant could be modelled as a first-order lag and a time delay. The performance of the more complex algorithms does not always, in practice, match that suggested by the theory. Leigh (1992, p. 185) summarises the position as follows:

1. PID algorithms perform surprisingly well in practice. They are robust and difficult to improve on significantly on a day-to-day basis unless very considerable effort is expended.
2. Feedforward and cascade algorithms perform well in those situations for which they were designed.
3. More complex algorithms tend to lose their supposed advantages once process parameters drift or noise begins to affect measurements.
4. In general, the use of a long sampling interval greatly increases the sensitivity of a control loop to process parameter drift. Complex algorithms

tend to use longer sampling intervals and hence are prone to parameter drift and noisy measurement problems.

The use of detailed plant models allows a wide variety of methods to be used in the design of the controller (see, for example, Astrom and Wittenmark, 1984; Franklin and Powell, 1980; Leigh, 1992; Katz, 1981; Kuo, 1980). Use of such methods gives rise to two types of representation for the controller:

- state-space representation of the difference equations; and
- transfer function in z^{-1}.

If the controller is in difference equation form it may be programmed directly; if it is given as a transfer function it has to be realised, that is converted either into an electronic (or other hardware) circuit or into a computer algorithm. There are four techniques for realisation:

- direct method 1;
- direct method 2;
- cascade; and
- parallel.

In terms of computer algorithms it can be shown that for a given quantisation limit (that is, wordlength) the cascade and parallel methods give algorithms in which the numerical errors are smaller than the errors in the algorithms produced by the two direct methods; for details see for example Katz (1981), Leigh (1992). In the next sections we deal briefly with the representation of the PID algorithm using the z-transform notation and with realisation approaches. If you do not have any control systems background you may wish to omit these sections and turn directly to section 4.11. If you want to try reading these sections but the z-transform is unfamiliar to you, you can for the purposes below treat z^{-1} as a delay operator. This means that writing ez^{-1} implies the value of e at the previous sampling interval. Hence

$$M(z) = (a + bz^{-1} + cz^{-2})E(z)$$

implies that, in the time domain, at the nth sample interval

$$m(n) = ae(n) + be(n-1) + ce(n-2)$$

or

$$m(nT) = ae(nT) + be[(n-1)T] + ce[(n-2)T]$$

where T is the sample interval.

4.10.1 The PID Controller in *Z*-transform Form

The PID controller can be expressed as a transfer function in z. Consider equation

4.4 and let $d = T_d/T_s$ and $g = T_s/T_i$. Then

$$m(n) = K_p\left(e(n) + g \sum_{k=1}^{n} e(k) + d[e(n) - e(n-1)]\right) \tag{4.23}$$

Since $D(z) = M(z)/E(z)$, $D(z)$ can be found by taking the z-transform of the right-hand side of equation 4.23 term by term to give

$$D(z) = K_p\left(1 + \frac{gz}{z-1} + d - dz^{-1}\right) \tag{4.24}$$

Equation 4.24 represents the parallel realisation of the PID controller and is shown in block form in Figure 4.11. Rewriting $K_p gz/(z-1)$ as $K_p gz/(1-z^{-1})$ the algorithm becomes

$$\begin{aligned}
x_1(i) &= K_p(1+d)e(i)\\
x_2(i) &= K_p ge(i) + x_2(i-1)\\
x_3(i) &= -K_p de(i-1)\\
m(i) &= x_1(i) + x_2(i) + x_3(i)
\end{aligned}$$

Substituting for d and g gives

$$\begin{aligned}
x_1(i) &= K_p(1 + T_d/T_s)e(i)\\
x_2(i) &= K_p(T_s/T_i)e(i) + x_2(i-1)\\
x_3(i) &= -K_p(T_d/T_s)e(i-1)
\end{aligned} \tag{4.25}$$

and it can be seen that equation 4.25 is the integral summation term which in equation 4.5 was expressed in the form $s(n) = s(n) + e(n)$. The algorithm from equation 4.25 is

$$\begin{aligned}
s(n) &= K_1 e(n) + s(n-1)\\
m(n) &= K_2 e(n) + K_3 e(n-1) + s(n)
\end{aligned}$$

where $K_1 = K_p T_s/T_i$, $K_2 = K_p(1 + T_d/T_s)$ and $K_3 = -K_p T_d/T_s$.

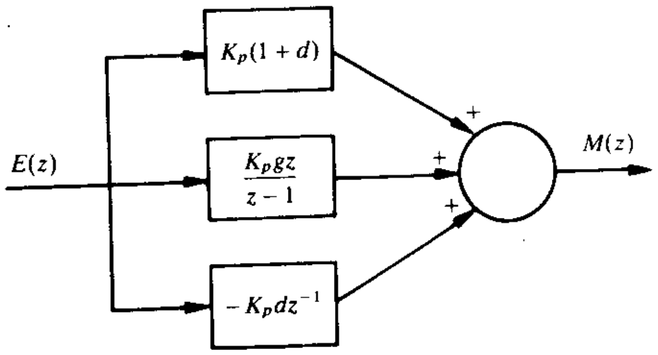

Figure 4.11 z-transform function form of PID controller.

Alternatively equation 4.24 can be rearranged to give

$$D(z) = K_p\left(\frac{(1+g+d)z^2 - (1+2d)z - d}{z(z-1)}\right) \tag{4.26}$$

Dividing the numerator and denominator by z^2 gives

$$D(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1}}$$

where

$$\begin{aligned} a_0 &= K_p(1+g+d) \\ a_1 &= -K_p(1+2d) \\ a_2 &= K_p d \\ b_1 &= -1 \end{aligned}$$

Direct implementation gives

$$m(i) = a_0 e(i) + a_1 e(i-1) + a_2 e(i-2) - b_1 m(i-1) \tag{4.27}$$

and substituting for a_0, a_1, a_2, b_1 gives

$$m(i) = K_p\left[\left(1 + \frac{T_s}{T_i} + \frac{T_d}{T_s}\right)e(i) - \left(1 + 2\frac{T_d}{T_s}\right)e(i-1) + \frac{T_d}{T_s}e(i-2) + m(i-1)\right] \tag{4.28}$$

which can easily be rearranged to give the velocity algorithm of equation 4.10.

4.10.2 Direct Method 1

The transfer function can be expressed as the ratio of two polynomials in z^{-1}:

$$\frac{M(z)}{E(z)} = D(z) = \frac{\Sigma_{j=0}^{n} a_j z^{-j}}{1 + \Sigma_{j=1}^{n} b_j z^{-j}} \tag{4.29}$$

The transfer function in equation 4.29 is converted directly into the difference equation

$$m_i = \sum_{j=0}^{n} a_j e_{i-j} - \sum_{j=1}^{n} b_j m_{i-j} \tag{4.30}$$

EXAMPLE 4.8

Consider a system with the transfer function

$$\frac{M(z)}{E(z)} = D(z) = \frac{3 + 3.6z^{-1} + 0.6z^{-2}}{1 + 0.1z^{-1} - 0.2z^{-2}} \tag{4.31}$$

Then by direct method 1 the computer algorithm is simply

$$m_i = 3e_i + 3.6e_{i-1} + 0.6e_{i-2} - 0.1m_{i-1} + 0.2m_{i-2}$$

4.10.3 Direct Method 2

Assuming as before that the transfer function can be expressed as in equation 4.29, then in direct method 2 the difference equation is formulated by introducing an auxiliary variable $P(z)$ such that

$$\frac{M(z)}{P(z)} = \sum_{j=1}^{n} a_j z^{-j} \tag{4.32}$$

and

$$\frac{P(z)}{E(z)} = \frac{1}{1 + \Sigma_{j=1}^{n} b_j z^{-j}} \tag{4.33}$$

From equations 4.32 and 4.33 two different equations are obtained:

$$m_i = \sum_{j=0}^{n} a_j p_{i-j} \tag{4.34}$$

and

$$p_i = e_i - \sum_{j=0}^{n} b_j p_{i-j} \tag{4.35}$$

Using the example above the following algorithm is obtained:

$$p_i = e_i - 0.1p_{i-1} + 0.2p_{i-2}$$
$$m_i = 3p_i + 3.6p_{i-1} + 0.6p_{i-2}$$

4.10.4 Cascade Realisation

If the transfer function is expressed as the product of simple block elements of first and second order as shown in Figure 4.12, then each element can be converted to a difference equation using direct method 1 and the overall algorithm is the set of

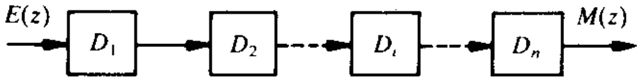

Figure 4.12 Cascade realisation.

difference equations. Equation 4.31 when expressed in this form becomes

$$\frac{M(z)}{E(z)}=D(z)=\frac{3(1+z^{-1})(1+0.2z^{-1})}{(1+0.5z^{-1})(1-0.4z^{-1})} \tag{4.36}$$

Hence

$$\begin{aligned} D_1 &= 3 \\ D_2 &= (1+z^{-1}) \\ D_3 &= (1+0.2z^{-1}) \\ D_4 &= 1/(1+0.5z^{-1}) \\ D_5 &= 1/(1-0.4z^{-1}) \end{aligned}$$

and letting x_1, x_2, x_3, x_4, and x_5 be the outputs of blocks D_1, D_2, D_3, D_4 and D_5 respectively, then

$$\begin{aligned} x_1(i) &= 3e(i) \\ x_2(i) &= x_1(i)+x_1(i-1) \\ x_3(i) &= x_2(i)+0.2x_2(i-1) \\ x_4(i) &= x_3(i)-0.5x_4(i-1) \\ x_5(i) &= x_4(i)+0.4x_5(i-1) \end{aligned}$$

4.10.5 Parallel Realisation

If the transfer function is expressed in fractional form or is expanded into partial fractions then it can be represented as shown in Figure 4.13. In this case each of the transfer functions D_1, D_2, and D_3 is expressed in difference equation form using

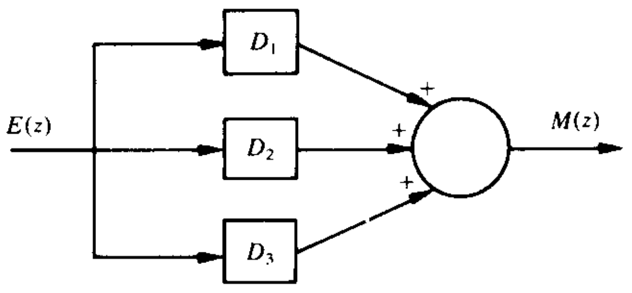

Figure 4.13 Parallel realisation.

direct method 1 and the output is obtained by summing the outputs from each block.

The partial fraction expansion of equation 4.31 is

$$\frac{M(z)}{E(z)} = D(z) = -3 - \frac{1}{1 + 0.5z^{-1}} + \frac{7}{1 - 0.4z^{-1}} \tag{4.37}$$

Hence $D_1 = -3$, $D_2 = -1/(1 + 0.5z^{-1})$, $D_3 = 7/(1 - 0.4z^{-1})$ and the algorithm is

$$\begin{aligned} x_1(i) &= -3e(i) \\ x_2(i) &= -e(i) - 0.5x_2(i-1) \\ x_3(i) &= 7e(i) + 0.4x_3(i-1) \\ m(i) &= x_1(i) + x_2(i) + x_3(i) \end{aligned}$$

4.10.6 Discretisation of Continuous Controllers

In this section we have assumed that a controller designed using a plant model has been designed using discrete design techniques. However, controllers can be designed using continuous system design methods and then discretised for digital implementation.

The problem of discretisation is interesting and considerably more complex than might at first be thought. There are a number of methods which can be used and these are summarised below. However, none of them exactly preserve the characteristics of the continuous system (time response, frequency response, pole-zero locations). The main methods are:

1. impulse invariant transform (z-transform);
2. impulse invariant transform withhold;
3. mapping of differentials;
4. bilinear (or Tustin) transform;
5. bilinear transform with frequency prewarping; and
6. mapping of poles and zeros (matched z-transform).

If it is not possible to give any firm indication of a best method for all applications, however, in general the bilinear transform (4 or 5) and pole-zero matching (6) give the closest approximations to the continuous system. Extensive comparisons of the methods can be found in Astrom and Wittenmark (1984), Franklin and Powell (1980), Leigh (1992) and Katz (1981).

4.11 SUMMARY

We have dealt at length with the implementation of a simple control algorithm in order to illustrate some of the practical problems that have to be overcome in implementing digital controllers. These problems apply whatever form of digital control algorithm is used. An important point to remember is that the more precise and accurate the algorithm the more precisely must the timing requirements be met. The standard PID algorithm, because it is not exactly matched to a particular plant, remains well behaved when there are variations in the sampling interval, and when samples are missed, algorithms that have been designed using discrete control design techniques or discretised forms of continuous algorithms are not always so well behaved and may be sensitive to small variations in sampling interval. For detailed consideration of these issues see Franklin and Powell (1980), Katz (1981) and Williamson (1991).

The reasons for the algorithm's widespread use are that it requires very little knowledge of the plant dynamics and the methods of determining the controller parameters are well known and understood (Auslander and Sagues, 1981; Ahson, 1983; Cohen and Coon, 1953; Leigh, 1992; Smith, 1972; Stephanopoulos, 1985; Takahashi *et al.*, 1970; Ziegler and Nichols, 1942). If knowledge of the plant model is available a wide variety of techniques can be used. The design can be carried out using continuous system design methods to find $G_c(s)$ followed by discretisation of $G_c(s)$ to give $G_c(z)$. A number of methods of mapping $G_c(s)$ to $G_c(z)$ are available (see section 4.10.6) but none give a controller $G_c(z)$ with exactly the same characteristics of $G_c(s)$. Care must be used since, for example, discretisation using first-order finite differences can easily result in changing a stable continuous-time system into an unstable discrete-time system. An alternative is to design $G_c(z)$ on the basis of a discrete-time model of the plant, $G_p(z)$, obtained either by discretisation of the continuous-time model $G_p(s)$ or by determining $G_p(z)$ directly.

In both cases we are concerned with how the controller is programmed, not with how it is designed (information on design techniques can be found in Franklin and Powell (1980), Iserman (1981), Katz (1981), Kuo (1980), Leigh (1992), and Smith (1972)).

Digital microcontrollers for PID control are now widely available from process instrument manufacturers; most of these units contain well-proven PID software and most incorporate some form of self-tuning or expert-system-based automatic tuning. Also available are software packages containing standard implementations of common control algorithms.

In most computer control applications the implementation of the direct digital control algorithms is a minor part of the system: the major efforts and complications arise in communicating with the plant, the operators, and other computers, and in providing a safe system that can handle alarm and fault conditions. In the rest of the book we turn our attention to the tools and techniques that help us in building complex systems in a safe and secure way.

EXERCISES

4.1 Many personal computers have interval timers, that is they have a counter which can be initialised and which is incremented (or decremented) at fixed intervals by an interrupt signal. Using the technique shown in Example 4.2 write a program to output the 'bell' character (07H) at a fixed interval (for example, 2 seconds). If you have access to a personal computer check the accuracy of the timing by using a stopwatch to time a number of rings.

4.2 A person's reaction time can be measured by sending, at random intervals, a character to a VDU screen and asking the subject to press a key when the character appears. If you have access to a personal computer or some other small computer write a program to carry out such an experiment.

4.3 Modify the code of Example 4.1 to incorporate the velocity subtraction method of preventing integral action wind-up. Assume that a logic signal is available to indicate when the control actuator is in saturation.

4.4 (a) Draw a flowchart to show how bumpless transfer (Method 2 – tracking of the manipulated variable) can be incorporated into the standard PID controller.
(b) Based on the flowchart write a program (in any language) for the system.

4.5 Write a program, in any language, to implement the velocity algorithm for the PID controller.

4.6 How would you incorporate (a) into the standard PID digital controller and (b) into the velocity form of the PID controller, the requirement that the manipulated variable should not change by more than 1% between two sample intervals?

4.7 Discuss the problems of testing the computer implementation of a digital control algorithm. Work out a test scheme which would minimise the amount of time required for test purposes on the actual plant. The scheme should show the various stages of testing and should be designed to eliminate coding errors and logic design errors prior to the connection of the controller to the plant.

4.8 The results of an open-loop response to a unit step input for a plant are:

Time (seconds)	*Output*
0.1	0.01
0.2	0.02
0.3	0.06
0.4	0.14
0.5	0.24
0.6	0.34
0.7	0.44
0.8	0.54
0.9	0.64
1.0	0.71
1.1	0.76
1.2	0.79
1.3	0.80

Find (a) the approximate plant model, (b) a suitable sampling interval for a digital PID controller and (c) estimates of the optimum controller settings for PI and PID control.

4.9 The analog system shown in Figure 4.14 can be discretised using the z-transform plus zero-order hold method. The resulting algorithm is

$$e(n) = r - c(n)$$
$$c(n) = (k/a^2)[Ae(n-1) + Be(n-2)] + [Cc(n-1) - Dc(n-2)]$$

where

$$A = T_s a - 1 + \exp(-aT_s)$$
$$B = 1 - \exp(-aT_s) - T_s a \exp(-aT_s)$$
$$C = 1 + \exp(-aT_s)$$
$$D = \exp(-aT_s)$$
$$T_s = \text{sampling interval}$$

Write a program which will enable you to calculate the change in output of the system, $c(n)$, with time. It is suggested that 50 values are calculated. The program should enable different values of k, a, T_s and r to be entered.

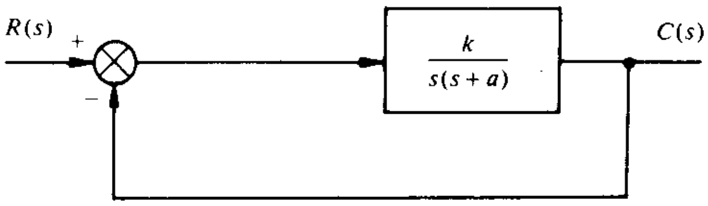

Figure 4.14 Control system for Exercise 4.9.

4.10 Using the program of Exercise 4.9, set $k = 2$, $a = 1$, $r = 1$ and investigate the response of the system for different values of T_s. It is suggested that $T_s = 0.02, 0.05, 0.1, 0.2, 0.5$. Compare the results (for example, in terms of maximum overshoot) with the exact solution for the continuous system (maximum overshoot = 30.5%).