REAL – TIME SYSTEMS

Module – 1

Introduction to Real – Time Systems:

Historical Background, RTS Definition, Classification of Real – Time Systems, Time constraints, Classification of programs.

Concepts of Computers Control:

Introduction, Sequence Control, Loop Control, Supervisory Control, Centralized Computer Control, Distributed System, Human-Computer interface, Benefits of Computer Control Systems.

Module - 2

Computer Hardware Requirements for RTS:

Introduction, General Purpose Computer, Single Chip Microcontroller, Specialized Processors, Process –Related Interfaces, Data Transfer Techniques, Communications, Standard Interface.

Module- 3

Languages For Real –Time Applications:

Introduction, Syntax Layout and Readability, Declaration and Initialization of Variables and Constants, Modularity and Variables, Compilation , Data Type, Control Structure, Exception Handling, Low –Level Facilities, Co routines, Interrupts and Device Handling, Concurrency, Real – Time Support, Overview of Real –Time Languages.

PART –B

Module-4

Operating Systems:

Introduction, Real –Time Multi –Tasking OS, Scheduling Strategies, Priority Structures, Task Management, Scheduler and Real –Time Clock Interrupt Handles, Memory Management ,Code Sharing, Resource control, Task Co-operation and Communication, Mutual Exclusion

Module-5

Design of RTSS General Introduction:

Introduction, Specification documentation, Preliminary design, Single –Program Approach, Foreground /Background, Multi- Tasking approach, Mutual Exclusion Monitors.

RTS Development Methodologies:

Introduction, Yourdon Methodology, Requirement definition For Drying Oven, Ward and Mellor Method, Hately and Pirbhai Method.

Text Books:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.

Reference: Books:

1. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.

2. **Embedded Systems**, Raj kamal, Tata MC Graw Hill, INDIA, 2005.

CONTENTS

MODULE – 1

Introduction to Real – Time Systems

Historical Background, RTS Definition, Classification of Real – Time Systems, Time constraints, Classification of programs.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.

Introduction to Real –Time Systems.

1.1 Historical Background:

The origin of the term Real –Time Computing is unclear. It was probably first used either with project whirlwind, a flight simulator developed by IBM for the U.S. Navy in 1947, or with SAGE, the Semiautomatic Ground Environment air defense system developed for the U.S. Air force in the early 1950s. Modern real-time systems, such as those that control Nuclear Power stations, Military Aircraft weapons systems, or Medical Monitoring Equipment, are complex and they exhibit characteristics of systems developed from the 1940s through the 1960s. Moreover, today's real time systems exist because the computer industry and systems requirements grew.

The earliest proposal to use a computer operating in real time as part of a control system was made in a paper by Brown and Campbell in 1950. It shows a computer in both the feedback and feed forward loops. The diagram is shown below:

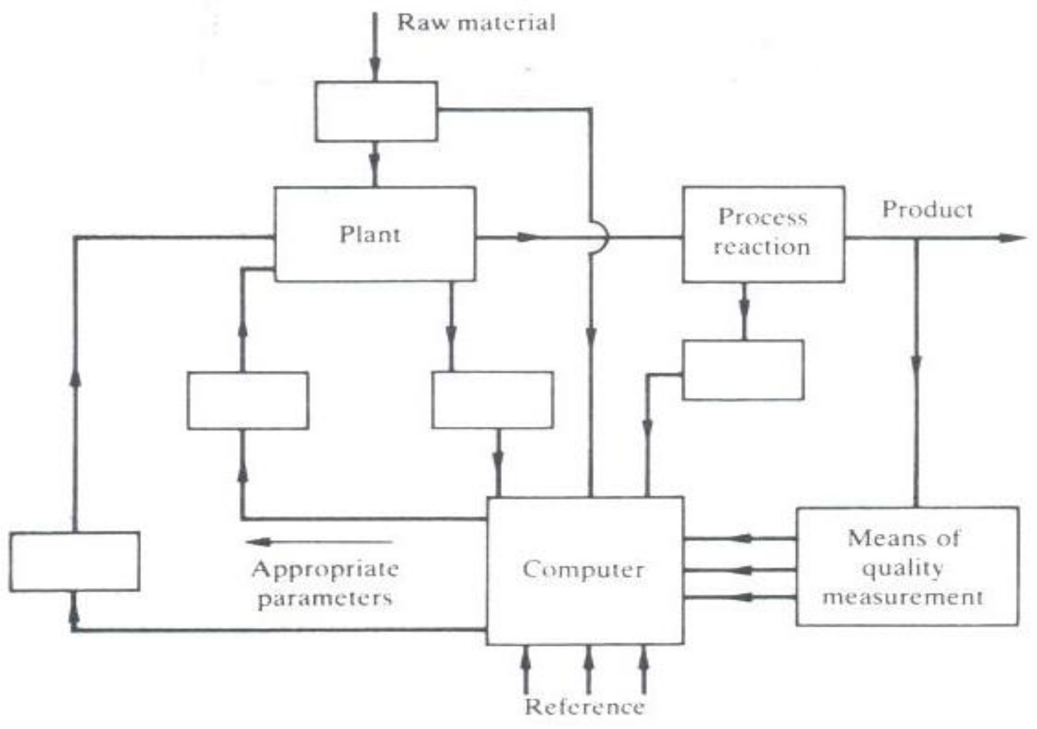


Figure: Computer used in control of plant.

The first digital Computers developed specifically for real time control were for airborne operation, and in 1954 a digitrac digital computer was successfully used to provide an automatic flight and weapons control system.

The application of digital computers to industrial control began in the late 1950s.

* The first industrial installation of a computer system was in September 1958. When the Louisiana Power and Light Company installed a Day Storm Computer system for plant monitoring at their power station in sterling, Louisiana.

* The first industrial Computer Control installation was made by the Texaco Company who installed an RW-300 (Ramo -Wooldridge Company) system at their Port Arthur refinery in Texas.

* During 1957-8 the Monsanto Chemical Company, in co-operation with the Ramo-Wooldridge company, studied the possibility of using computer control and in October 1958 decided to implement a scheme on the ammonia plant at luling, Louisiana.

* The same system was installed by the B.F. Goodrich Company on their acrylanite plant at Calvert City, Kentucky, in 1959-60.

* The first direct digital control (DDC) computer system was the Ferranti Argus 200 system installed in November 1962 at the ICI ammonia – soda plant at Fleetwood Lancashire.

The computers used in the early 1960s combined magnetic core memories and drum stores, the drum eventually giving way to hard disk drives. They included the General Electric 4000 series, IBM 1800, CDC 1700, Foxboro Fox 1 and 1A, the SDS and Xerox SIGMA Series, Ferranti Argus and Elliot Automation 900 series. The attempt to resolve some of the problems of the early machines led to an increase in the cost of systems.

The consequence of the generation of further problems particularly in the development of the software. The increase in the size of the programs meant that not all the code could be stored in core memory; provision to be made for the swapping of code between the drum memory and core. The solution appeared to lie in the development general purpose real-time operating systems and high –level languages.

In the late 1960s real time operating system were developed and various PROCESS FORTRAN Compilers made their appearance. The problems and the costs involved in attempting to do everything in one computer led users to retreat to smaller system for which the newly developing minicomputer (DEC PDP-8, PDP-11, Data General Nova, Honey well 316 etc.) was to prove ideally suited.

1.2 REAL-TIME SYSTEM DEFINITION:

Real- time processing normally requires both parallel activities and fast response. In fact, the term ‘real –time’ is often used synonymously with ‘multi – tasking’ or ‘multi- threading’.

Although there is no clear dividing line between real-time and non-real-time Systems, there are a set of distinguishing features:

The oxford Dictionary of computing offers the definition:

Any system in which the time at which the output is produced is significant. This usually because the input corresponded to some movement in the physical world, and output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

This definition covers a very wide range of systems; for examples, from workstations running under the UNIX operating system from which the user expects to receive a response within a few seconds through to aircraft engine control systems which must respond within a specified time and failure to do so could cause the loss of control and possibly the loss of many lives.

Latter type of system cooling (1991) offers the definition:

Real- time systems are those which must produce correct responses within a definite time limit.

An alternative definition is:

A real- time system read inputs from the plant and sends control signals to the plant at times determined by plant operational considerations.

We can therefore define a real –time program as:

A program for which the correctness of operation depends both on the logical results of the computation and the time at which the results are produced. One of the classification schema to identify real-time.

Timing:

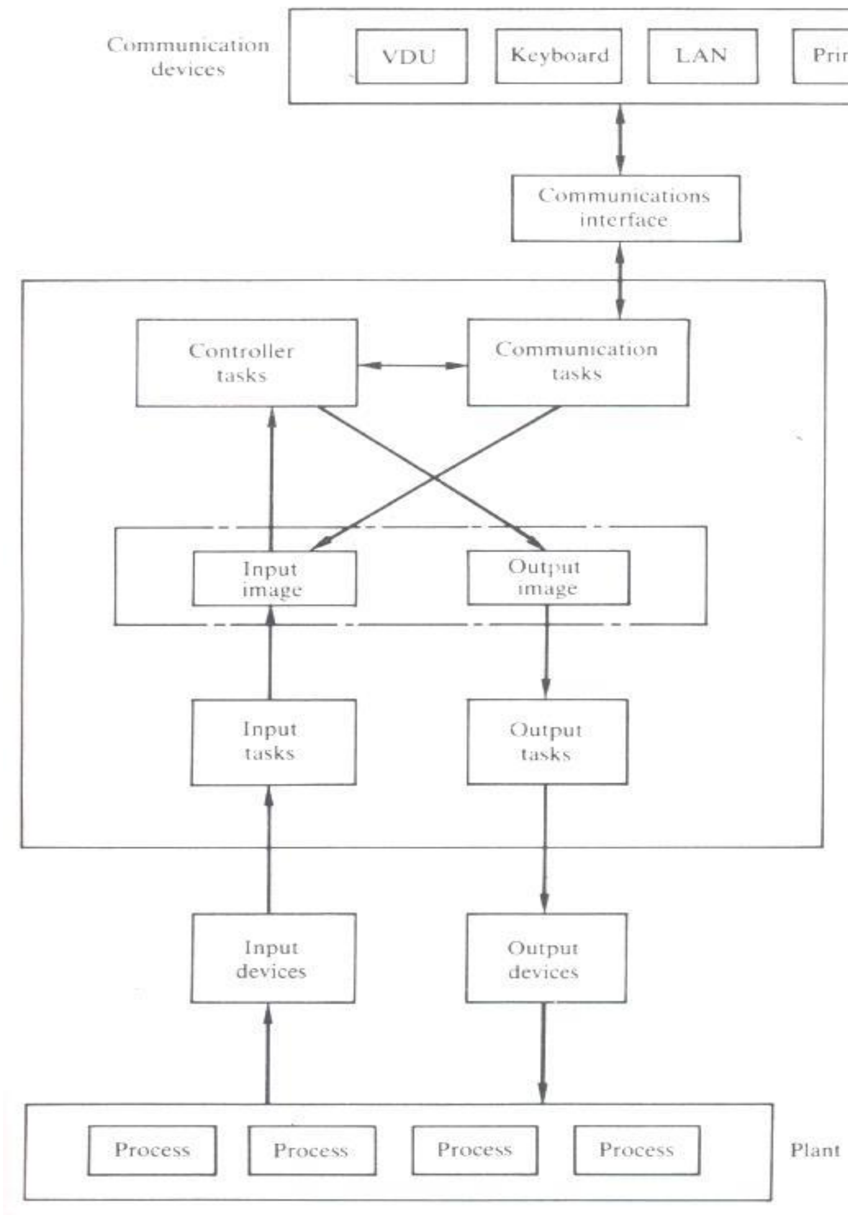
The most common definition of a real-time system involves a statement similar to 'Real-time systems are required to compute and deliver correct results within a specified period of time'. Does this mean that a non-real-time system such as a payroll program, could point salary cheques two years late, and be forgiven because it was not a real-time system? Hardly so! Obviously there are time constraints on non-real-time systems too.

1.3 CLASSIFICATION OF REAL-TIME SYSTEM

A common feature of real-time systems and embedded computers is that the computer is connected to the environment within which it is working by a wide range of interface device and receives and sends a variety of stimuli. For example, the plant input, plant output, and communication tasks shown in figure:

In below figure one common feature they are connected by physical devices to processor which are external to computer. External processes all operate in their own time-scale and the computer is said to operate in real time if actions carried out in the computer relate to the time –scales of the external processes. Synchronization between the external processes and the internal actions (tasks) carried out by the computer may be defined in term of the passage of time, or the actual time of day, in which case the system is said to be clock based the operations are carried out according to a time schedule.

Other category, interactive, in which the relationship between the actions in the computer and the system is much more loosely defined. The control tasks, although not obviously and directly connected to the external environment, also need to operate in real -time, since time is usually involved in determining the parameters of the algorithms.



1.3.1 CLOCK – BASED TASKS (CYCLIC, PERIODIC):

Clock – based tasks are typically referred to as cyclic or periodic tasks where the terms can imply either that the task is to run once per time period T (or cycle time T), or is to run at exactly T unit intervals.

The completion of the operations within the specified time is dependent on the number of operations to be performed and the speed of the computer.

Synchronization is usually obtained by adding a clock to the computer system- referred as a real-time clock that uses signal from this clock to interrupt the operations of the computer at some predetermined fixed time interval.

For example in process plant operation, the computer may carry out the plant input, plant output and control tasks in response to the clock interrupt or, if the clock interrupt has been set at a faster rate than the sampling rate, it may count each interrupt until it is time to run the tasks.

In larger system the tasks may be subdivided into groups for controlling different parts of the plant and these may need to run a different sampling rate. A tasks or process comprises some code, its associated data and a control block data structure which the operating system uses to define and manipulate the task.

1.3.2 EVENT – BASED TASKS (APERIODIC):

Events occurring at non-deterministic interval and event-based tasks are frequently referred as aperiodic tasks. Such tasks may have deadlines expressed in term of having start times or finish times (or even both).

For example, a task may be required to start within 0.5 seconds of an event occurring, or alternatively it may have to produce an output (end time) within 0.5 seconds of the events. For many system where actions have to be performed not at particular times or time intervals but in response to some event.

Examples: Turning off a pump or closing a value when the level of a liquid in a tank reaches a predetermined valve; or switching a motor off in response to the closure of a micro switch indicating that some desired position has been reached.

Event based systems are also used extensively to indicate alarm conditions and initiate alarm actions.

1.3.3 INTERACTIVE SYSTEM:

Interactive systems probably represent the largest class of real-time systems and cover such systems as automatic bank tellers; reservation systems for hotels, airlines and car rental companies; computerized tills, etc. The real-time requirement is usually expressed in terms such as 'the average response time must not exceed ... ‘

For example, an automatic bank teller system might require an average response time not exceeding 20 seconds. Superficially this type of system seems similar to the event-based system in that it apparently responds to a signal from the plant (in this case usually a person), but it is different because it responds at a time determined by the internal state of the computer and without any reference to the environment. An automatic bank teller does not know that you are about to miss a train, or that it is raining hard and you are getting wet: its response depends on how busy the communication lines and central computers are (and of course the *wire* of your account).

Many interactive systems give the impression that they are clock based in that they are capable of displaying the date and time; they do indeed have a real-time clock which enables them to keep track of time.

1.4 TIME CONSTRAINTS:

Real time systems are divided into two categories:

- Hard real-time: these are systems that must satisfy the deadlines on each and every occasion.
- Soft real-time: these are systems for which an occasional failure to meet a deadline does not comprise the correctness of the system.

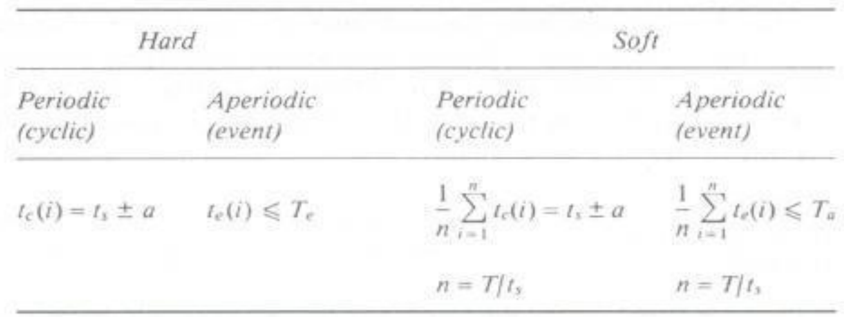
A typical example of a hard real-time control system is the temperature control loop of the hot-air blower system described above. In control terms, the temperature loop is a *sampled data* system. Design, of a suitable control algorithm for this system involves the choice of the sampling interval Ts. If we assume that a suitable sampling interval is 10 ms, then at 10 ms intervals the input value must be read, the control calculation carried out and the output value calculated, and the output value sent to the heater drive.

As an example of hard time constraints associated with event-based tasks let us assume that the hot-air blower is being used to dry a component which will be damaged if exposed to temperatures greater than 50°C for more than 10 seconds. Allowing for the time taken for the air to travel from the heater to the outlet and the cooling time of the heater element - and for a margin of safety - the alarm response requirement may be, say, that overt temperature should be detected and the heater switched off within seven seconds of the over temperature occurring. The general form of this type of constraint is that the computer must respond to the event within some specified maximum time.

An automatic bank teller provides an example of a system with a soft time constraint. A typical system is event initiated in that it is started by the customer placing their card in the machine. The time constraint on the machine responding will be specified in terms of an average response time of, say, 10 seconds, with the average being measured over a 24 hour period. (Note that if the system has been carefully specified there will also be a maximum time; say 30 seconds, within which the system should respond.) The actual response time will vary: if you have used such a system you will have learned that the response time obtained between 12 and 2 p.m. on a Friday is very different from that at 10 a.m. on a Sunday.

A hard time constraint obviously represents a much more severe constraint on the performance of the system than a soft time constraint and such systems present a difficult challenge both to hardware and to software designers. Most real-time systems contain a mixture of activities that can be classified as clock based, event based, and interactive with both hard and soft time constraints (they will also contain activities which are not real time). A system designer will attempt to reduce the number of activities (tasks) that are subject to a hard time constraint.

Formally the constraint is defined as follows:



Hard		*Soft*	
Periodic (cyclic)	*Aperiodic (event)*	*Periodic (cyclic)*	*Aperiodic (event)*
$t_c(i) = t_s \pm a$	$t_e(i) \leqslant T_e$	$\frac{1}{n}\sum_{i=1}^{n} t_c(i) = t_s \pm a$	$\frac{1}{n}\sum_{i=1}^{n} t_e(i) \leqslant T_a$
		$n = T/t_s$	$n = T/t_s$

tc (i) the interval between the i and i-I cycles,

te (i) the response time to the ith occurrence of event e,

ts the desired periodic (cyclic) interval,

Te	the maximum permitted response time to event e,
Ta	the average permitted response time to event e measured over some time interval T,
n	the number of occurrences of event e within the time interval T, or the number of cyclic repetitions during the time interval T,
a	a small timing tolerance.

For some systems and tasks the timing constraints may be combined in some form or other, or relaxed in some way.

1.5 CLASSIFICATION OF PROGRAMS:

The importance of separating the various activities carried out by computer control systems into real-time and non-real-time tasks, and in subdividing real-time tasks into the two different types, arises from the different levels of difficulty of designing and implementing the different types of computer program. Experimental studies have shown clearly that certain types of program, particularly those involving real time and interface operations, are substantially more difficult to construct than, for instance, standard data processing programs (Shooman, 1983; Pressman, 1992).The division of software into small, coherent *modules* is an important design technique and one of the guidelines for module division that we introduce is to put activities with different types of time constraints into separate modules.

Theoretical work on mathematical techniques for proving the correctness of a program, and the development of formal specification languages, such as 'z' and VOM, has clarified the understanding of differences between different types of program. Pyle (1979), drawing on the work of Wirth (1977), presented definitions identifying three types of programming:

- Sequential;
- Multi-tasking; and
- Real-time.

The definitions are based on the kind of arguments which would have to be made in order to verify, that is to develop a formal proof of correctness for programs of each type.

1.5.1 SEQUENTIAL:

In classical sequential programming *actions* are strictly ordered as a time sequence: the behavior of the program depends only on the effects of the individual *actions* and their *order;* the time taken to perform the action is not of consequence. Verification, therefore, requires two kinds of argument:

1. That a particular statement defines a stated action; and
2. That the various program structures produce a stated sequence of events.

1.5.2 MULTI-TASKING:

A multi-task program differs from the classical sequential program in that the actions it is required to perform are not necessarily disjoint in time; it may be necessary for several actions to be performed in parallel. Note that the *sequential relationships* between the actions may still be important. Such a program may be built from a number of parts (processes or tasks are the names used for the parts) which are themselves partly sequential, but which are executed concurrently and which communicate through shared variables and synchronization signals.

Verification requires the application of arguments for sequential programs with some additions. The task (processes) can be verified separately only if the constituent variables of each task (process) are distinct. If the variables are shared, the potential concurrency makes the effect of the program unpredictable (and hence not capable of verification) unless there is some further rule that governs the sequencing of the several actions of the tasks (processes). The task can proceed at any speed: the correctness depends on the actions of the synchronizing procedure.

1.5.3 REAL-TIME:

A real-time program differs from the previous types in that, in addition to its actions not necessarily being disjoint in time, the sequence of some of its actions is not determined by the designer but by the environment - that is, by events occurring in the outside world which occur in real time and without reference to the internal operations of the computer. Such events cannot be made to conform to the intertask synchronization rules.

A real-time program can still be divided into a number of tasks but communication between the tasks cannot necessarily wait for a synchronization signal: the environment task cannot be delayed. (Note that in process control applications the main environment task is usually that of keeping real time, that is a real-time clock task. It is this task which provides the timing for the scanning tasks which gather information from the outside world about the process.) In real-time programs, in contrast to the two previous types of program, the *actual time taken* by an action is an essential factor in the process of verification. We shall assume that we are concerned with real-time software and references to sequential and multi-tasking programs should be taken to imply that the program is real time. Non-real-time programs will be referred to as standard program.

Consideration of the types of reasoning necessary for the verification of programs is important, not because we, as engineers, are seeking a method of formal proof, but because we are seeking to understand the factors which need to be considered when designing real-time software. Experience shows that the design of real-time software is significantly more difficult than the design of sequential software. The problems of real-time software design have not been helped by the fact that the early high-level languages were sequential in nature and they did not allow direct access to many of the detailed features of the computer hardware.

As a consequence, real-time features had to be built into the operating system which was written in the assembly language of the machine by teams of specialist programmers. The cost of producing such operating systems was high and they had therefore to be general purpose so that they could be used in a wide range of applications in order to reduce the unit cost of producing them. These operating systems could be *tailored,* that is they could be reassembled to exclude or include certain features, for example to change the number of tasks which could be handled, or to change the number of input/output devices and types of device. Such changes could usually only be made by the supplier.

Excepted Question:

1. Explain the difference between a real-time program and a non-real-time program.
   Why are real-time programs more difficult to verify than non-real-time programs?
2. To design a computer-based system to control all the operations of a retail petrol (gasoline) station (control of pumps, cash receipts, sales figures, deliveries, etc.).
   What type of real-time system would you expect to use?
3. Classify any of the following systems as real-time?
   In each case give reasons for your answer and classify the real-time systems as hard or soft.
   (a) A simulation program run by an engineer on a personal computer.

(b) An airline seat-reservation system with on-line terminals.

(c) A microprocessor-based automobile ignition and fuel injection system.

(d) A computer system used to obtain and record measurements of force and strain from a tensile strength testing machine.

e) An aircraft autopilot.

4 An automatic bank teller works by polling each teller in turn. Some tellers are located outside buildings and others inside. How the polling system could be organized to ensure that the waiting time at the outside locations was less than at the inside locations?

5 .Explain the precision required for the analog-to-digital and digital-to-analog converters taking hot-air blower as an example?

MODULE– 1

Concepts of Computers Control

Introduction, Sequence Control, Loop Control, Supervisory Control, Centralized Computer Control, Distributed System, Human-Computer interface, Benefits of Computer Control Systems.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.

2.1 Concepts of computers control:

Introduction:

Computers are now used in so many different ways that we could take it up by simply describing various applications. However, when we examine the applications closely we find that there are many common features. The basic features of computer control systems are illustrated in the following sections using examples drawn from industrial process control. In this field applications are typically classified under the following headings:

• Batch;

• Continuous; and

• Laboratory (or test).

The categories are not mutually exclusive: a particular process may involve activities which fall into more than one of the above categories; they are, however, useful for describing the general character of a particular process.

BATCH:

The term *batch* is used to describe processes in which a sequence of operations are carried out to produce a quantity of a product - the batch - and in which the sequence is then repeated to produce further batches. The specification of the product or the exact composition may be changed between the different runs.

A typical example of batch production is rolling of sheet steel. An ingot is passed through the rolling mill to produce a particular gauge of steel; the next ingot may be either of a different composition or rolled to a different thickness and hence will require different settings of the rolling mill.

An important measure in batch production is *set-up* time (or *change-over* time), that is, the time taken to prepare the equipment for the next production batch. This is *wasted* time in that no output is being produced; the ratio between *operation* time (the time during which the product is being produced) and set-up time is important in determining a suitable batch size.

In mechanical production the advent of the NC (Numerically Controlled) machine tool which can be set up in a much shorter time than the earlier automatic machine tool has led to a reduction in the size of batch considered to be economic.

CONTINUOUS:

The term *continuous* is used for systems in which production is maintained for long periods of time without interruption, typically over several months or even years. An example of a continuous system is the catalytic cracking of oil in which the crude oil enters at one end and the various products - fractionates – are removed as the process continues. The ratio of the different fractions can be changed but this is done without halting the process.

Continuous systems may produce batches, in that the product composition may be changed from time to time, but they are still classified as continuous since the change in composition is made without halting the production process.

A problem which occurs in continuous processes is that during change-over from one specification to the next, the output of the plant is often not within the product tolerance and must be scrapped. Hence it is financially important that the change be made as quickly and smoothly as possible. There is a trend to convert processes to continuous operation - or, if the whole process cannot be converted, part of the process.

For example, in the baking industry bread dough is produced in batches but continuous ovens are frequently used to bake it whereby the loaves are placed on a conveyor which moves slowly through the oven. An important problem in mixed mode systems, that is systems in which batches are produced on a continuous basis, is the tracking of material through the process; it is obviously necessary to be able to identify a particular batch at all times.

LABORATORY SYSTEMS:

Laboratory-based systems are frequently of the operator-initiated type in that the computer is used to control some complex experimental test or some complex equipment used for routine testing. A typical example is the control and analysis of data from a vapour phase chromatograph.

Another example is the testing of an audiometer, a device used to lest hearing. The audiometer has to produce sound levels at different frequencies; it is complex in that the actual level produced is a function of frequency since the sensitivity of the human ear varies with frequency. Each audiometer has to be tested against a sound-level meter and a test certificate produced. This is done by using a sound-level meter connected to a computer and using the output from the computer to drive the audiometer through its frequency range. The results printed out from the test computer provide the test certificate.

As with attempts to classify systems as batch or continuous so it can be difficult at times to classify systems solely as laboratory. The production of steel using the electric arc furnace involves complex calculations to determine the appropriate mix of scrap, raw materials and alloying additives. As the melt progresses samples of the steel are taken and analyzed using a spectrometer. Typically this instrument is connected to a computer which analyses the results and calculates the necessary adjustment to the additives. The computer used may well be the computer which is controlling the arc furnace itself.

In whatever way the application is classified the activities being carried out will include:

- Data acquisition;
- Sequence control;
- Loop control (DDC);
- Supervisory control;
- Data analysis;
- Data storage; and

• Human-computer interfacing (HCI).

• Efficiency of operation;

• Ease of operation;

• Safety;

•Improved products;

• Reduction in waste;

• Reduced environmental impact; and

• A reduction in direct labour.

GENERAL EMBEDDED SYSTEMS:

In the general range of systems which use embedded computers – from domestic appliances, through hi-fi systems, automobile management systems, intelligent instruments, active control of structures, to large flexible manufacturing systems and aircraft control systems - we will find that the activities that are carried out in the computer and the objectives of using a computer are similar to those listed above. The major differences will lie in the balance between the different activities, the time-scales involved, and the emphasis given to the various objectives.

2.2 SEQUENCE CONTROL:

Although sequence control occurs in some part of most systems it often predominates in batch systems and hence a batch system is used to illustrate it. Batch systems are widely used in the food processing and chemical industries where the operations carried out frequently involve mixing raw materials, carrying out some process, and then discharging the product. A typical reactor vessel for this purpose is shown in Figure 2.1 below.

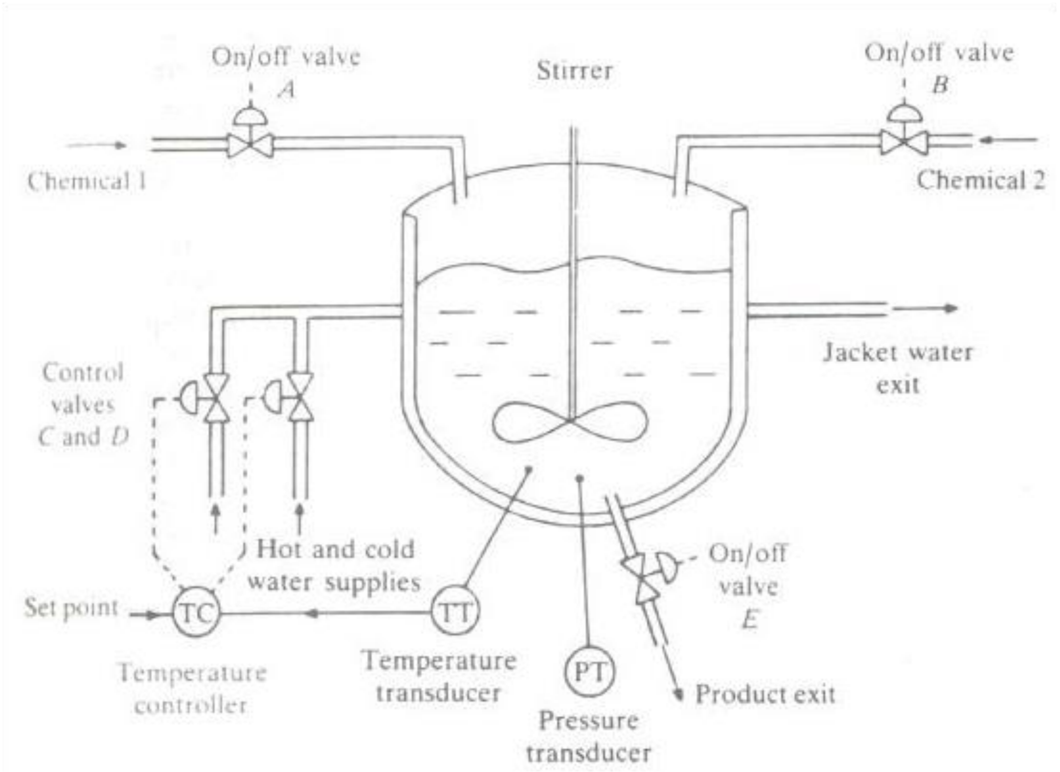


Figure2.1: A simple chemical reactor vessel

A chemical is produced by the reaction of two other chemicals at a specified temperature. The chemicals are mixed together in a sealed vessel (the reactor) and the temperature of the reaction is controlled by feeding hot or cold water through the water jacket which surrounds the vessel.

The water flow is controlled by adjusting valves C and D. The flow of material into and out of the vessel is regulated by the valves A, Band E. The temperature of the contents of the vessel and the pressure in the vessel are monitored.

The procedure for the operation of the system may be as follows:

1. Open valve A to charge the vessel with chemical 1.
2. Check the level of the chemical in the vessel (by monitoring the pressure in the vessel); when the correct amount of chemical has been admitted, close valve A.
3. Start the stirrer to mix the chemicals together.
4. Repeat stages 1 and 2 with valve B in order to admit the second chemical.
5. Switch on the three-term controller and supply a set point so that the chemical mix is heated up to the required reaction temperature.
6. Monitor the reaction temperature; when it reaches the set point, start a timer to

time the duration of the reaction.

7. When the timer indicates that the reaction is complete, switch off the controller and open valve C to cool down the reactor contents. Switch off the stirrer.
8. Monitor the temperature; when the contents have cooled, open valve E to remove the product from the reactor.

When implemented by computer all of the above actions and timings would be based upon software. For a large chemical plant such sequences can become very lengthy and intricate and, to ensure efficient operating, several sequences may take place in parallel.

The processes carried out in the single reactor vessel shown in Figure 2.1 are often only part of a larger process as is shown in Figure 2.2. In this plant two reactor vessels (R 1 and R2) are used alternately, so that the processes of preparing for the next batch and cleaning up after a batch can be carried out in parallel with the actual production. Assuming that R 1 has been filled with the mixture and the catalyst, and the reaction is in progress, there will be for R 1: loop control of the temperature and pressure; operation of the stirrer; and timing of the reaction (and possibly some in process measurement to determine the state of the reaction). In parallel with this, vessel R2 will be cleaned - the wash down sequence - and the next batch of raw material will be measured and mixed in the mixing tank.

Meanwhile, the previous batch will be thinned down and transferred to the appropriate storage tank and, if there is to be a change of product or a change in product quality, the thin-down tank will be cleaned. Once this is done the next batch can be loaded into R2 and then, assuming that the reaction in R1 is complete, the contents of R1 will be transferred to the thin-down tank and the wash down procedure for R1 initiated. The various sequences of operations required can become complex and there may also be complex decisions to be made as to when to begin a sequence. The sequence initiation may be left to a human operator or the computer may be programmed to supervise the operations (supervisory control - see below). The decision to use human or computer supervision is often very difficult to make.

The aim is usually to minimize the time during which the reaction vessels are idle since this is unproductive time. The calculations needed and the evaluation of options can be complex, particularly if, for example, reaction times vary with product mix, and therefore it would be expected that decisions made using computer supervisory control would give the best results. however, it is difficult using computer control to obtain the same flexibility that can be achieved using a human operator (and to match the ingenuity of good operators). As a consequence many supervisory systems are mixed; the computer is programmed to carry out the necessary supervisory calculations and to present its decisions for confirmation or rejection by the operator, or alternatively it presents a range of options to the operator.

In most batch systems there is also, in addition to the sequence control, some continuous feedback control: for example, control of temperatures, pressures, flows, speeds or currents. In process control terminology continuous feedback control is referred to as loop control or modulating control and in modern systems this would be carried out using DOC.

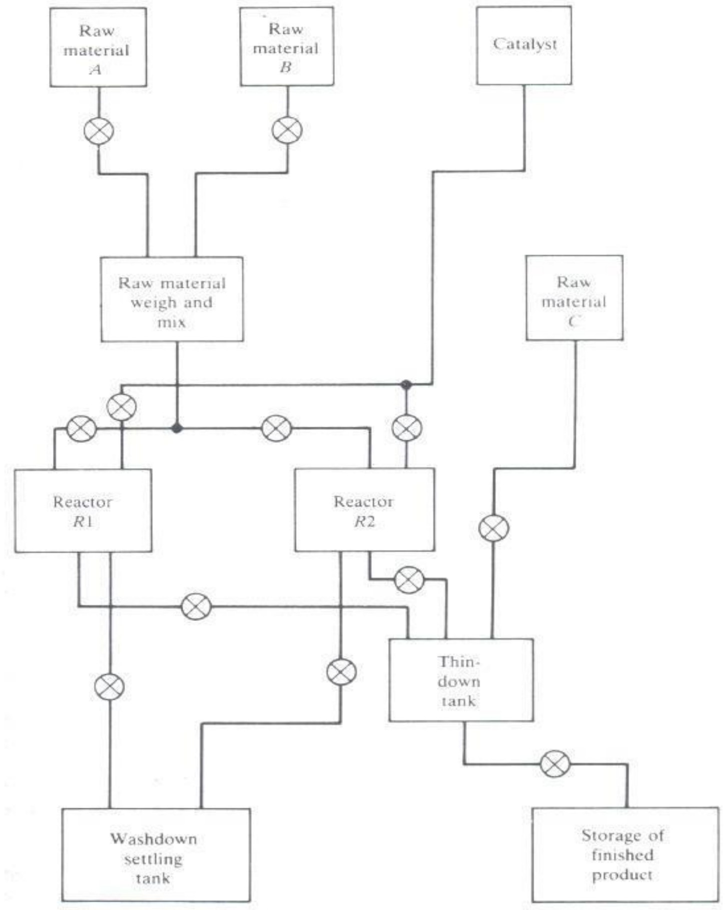


Figure2.2: Typical chemical batch process.

A similar mixture of sequence, loop and supervisory control can be found in continuous systems. Consider the float glass process shown in Figure 2.3. The raw material - sand, powdered glass and fluxes (the frit) - is mixed in batches and fed into the furnace. It melts rapidly to form a molten mixture which flows through the furnace. As the molten glass moves through the furnace it is refined. The process requires accurate control of temperature in order to maintain quality and to keep fuel costs to a minimum - heating the furnace to a higher temperature than is necessary wastes energy and increases costs. The molten glass flows out of the furnace and forms a ribbon on the float bath; again, temperature control is important as the glass ribbon has to cool sufficiently so that it can pass over rollers without damaging its surface.

The continuous ribbon passes into the lehr where it is annealed and where temperature control is again required. It then passes under the cutters which cut it into sheets of the required size; automatic stackers then lift the sheets from the production line. The whole of this process is controlled by several computers and involves loop, sequence and supervisory control. Sequence control systems can vary from the large - the start-up of a large boiler turbine unit in a power station when some 20000 operations and checks may have to be made - to the small - starting a domestic washing machine. Most sequence control systems are simple and frequently have no loop control. They are systems which in the past would have been controlled by relays, discrete logic, or integrated circuit logic units. Examples are simple presses where the sequence might be: locate blank, spray lubricant, lower press, raise press, remove article, spray lubricant. special computer systems known as *programmable logic controllers* (PLCs).

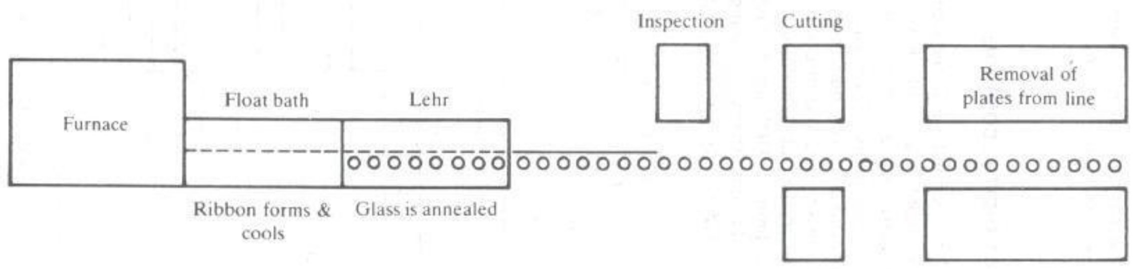


Figure 2.3: Schematic of float glass process.

2.3 LOOP CONTROL (DIRECT DIGITIAL CONTROL):

In direct digital control (DDC) the computer is in the feedback loop as is shown in Figure 2.4., the system shown in Figure 2.4 is assumed to involve several control loops all of which are handled within one computer.

A consequence of the computer being in the feedback loop is that it forms a *critical* component in terms of the reliability of the system and hence great care is needed to ensure that, in the event of the failure or malfunctioning of the computer, the plant remains in a safe condition. The usual means of ensuring safety are to limit the DDC unit to making *incremental* changes to the actuators on the plant; and to limit the rate of change of the actuator settings (the actuators are labeled *A* in Figure 2.4).

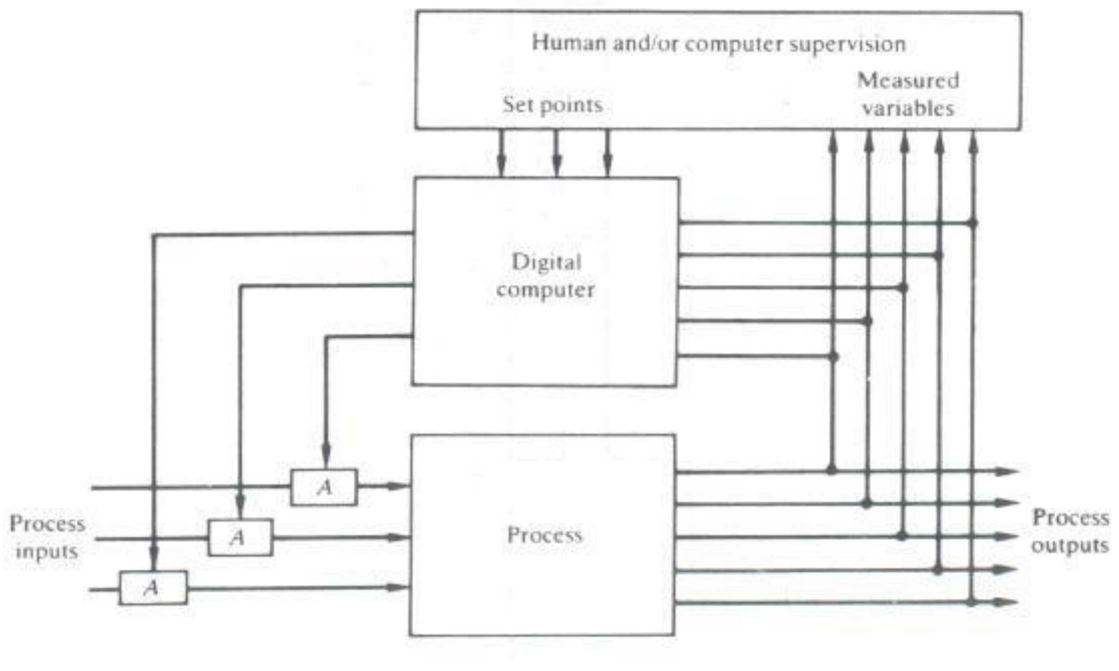


Figure 2.4: Direct digital control.

The advantages claimed for DDC over analog control are:

1. Cost - a single digital computer can control a large number of loops. In the early days the break-even point was between 50 and 100 loops, but now with the introduction of microprocessors a single-loop DDC unit can be cheaper than an analog unit.

2. Performance - digital control offers simpler implementation of a wide range of control algorithms, improved controller accuracy and reduced drift.

3. Safety - modern digital hardware is highly reliable with long mean-time between- failures and hence can improve the safety of systems. However, the software used in programmable digital systems may be much less reliable than the hardware.

The development of integrated circuits and the microprocessor have ensured that in terms of cost the digital solution is now cheaper than the analog. Single-loop controllers used as stand-alone controllers are now based on the use of digital techniques and contain one or more microprocessor chips which are used to implement DDC algorithms. The adoption of improved control algorithms has, however, been slow. Many computer control implementations have simply taken over the well-established analog PID (Proportional + Integral + Derivative) algorithm.

PID CONTROL:

The PID control algorithm has the general form

$$m(t) = Kp\ [e(t) + 1/Ti \int_0^1 e(t)dt + T_d\ de(t)/dt]$$

Where $e(t) = r(t) - c(t)$ and $c(t)$ is the measured variable, $r(i)$ is reference value or set point, and $e(t)$ is error; Kp is the overall controller gain; T; is the integral action time; and T_d is the derivative action time.

For a wide range of industrial processes it is difficult to improve on the control performance that can be obtained by using either PI or PID control (except at considerable expense) or it is for this reason that the algorithms are widely used. For the majority of systems PI control is all that is necessary. Using a control signal that is made proportional to the error between the desired value of an output and the actual value of the output is an obvious and (hopefully) a reasonable strategy. Choosing the value of Kp involves a compromise: a high value of Kp gives a small steady-state error and a fast response, but the response will be oscillatory and may be unacceptable in many applications; a low value gives a slow response and a large steady-state error. By adding the integral action term the steady-state error can be reduced to zero since the integral term, as its name implies, integrates the error signal with respect to time. For a given error value the rate at which the integral term increases is determined by the integral action time Ti. The major advantage of incorporating an integral term arises from the fact that it compensates for changes that occur in the process being controlled.

A purely proportional controller operates correctly only under one particular set of process conditions: changes in the load on the process or some environmental condition will result in a steady-state error; the integral term compensates for these changes and reduces the error to zero. For a few processes which are subjected to sudden disturbances the addition of the derivative term can give improved performance. Because derivative action produces a control signal that is related to the rate of change of the error signal, it anticipates the error and hence acts to reduce the error that would otherwise arise from the disturbance.

In fact, because the PID controller copes perfectly adequately with 90070 of all control problems, it provides a strong deterrent to the adoption of new control system design techniques. DDC may be applied either to a single-loop system implemented on a small microprocessor or to a large system involving several hundred loops. The loops may be cascaded, that is with the output or actuation signal of one loop acting as the set point for another loop, signals may be added together (ratio loops) and conditional switches may be used to alter signal connections.

A typical industrial system is shown in Figure 2.5. This is a steam boiler control system. The steam pressure is controlled by regulating the supply of fuel oil to the burner, but in order to comply with the pollution regulations a particular mix of air and fuel is required. We are not concerned with how this is achieved but with the elements which are required to implement the chosen control system.

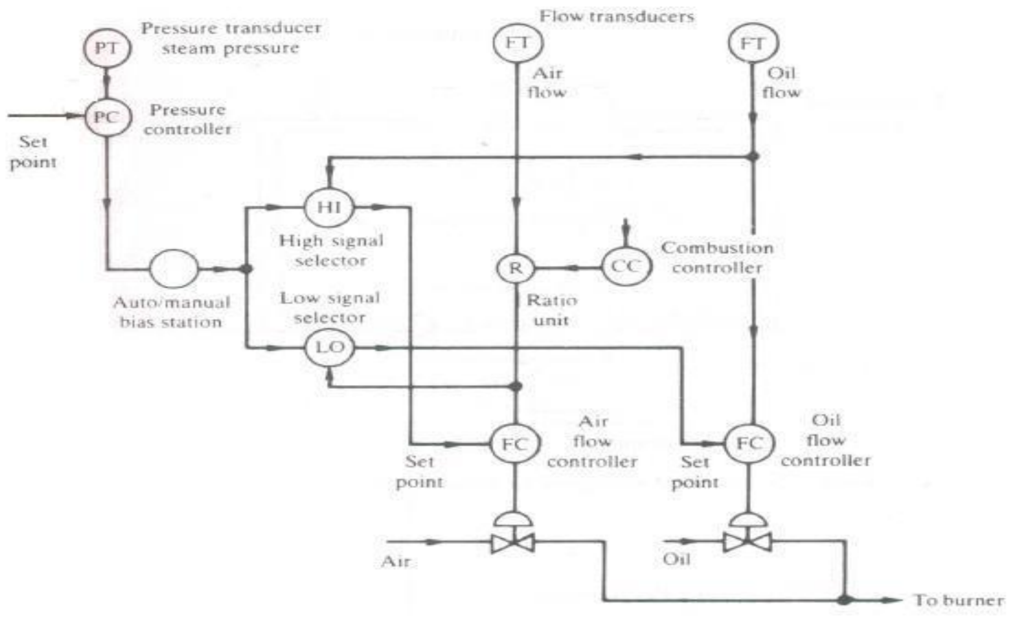


Figure 2.5: A boiler control scheme.

DDC APPLICATIONS:

DDC may be applied either to a single-loop system implemented on a small microprocessor or to a large system involving several hundred loops. The loops may be cascaded, that is with the output or actuation signal of one loop acting as the set point for another loop, signals may be added together (ratio loops) and conditional switches may be used to alter signal connections. A typical industrial system is shown in Figure 2.5. This is a steam boiler control system.

The steam pressure control system generates an actuation signal which is fed to an auto/manual bias station. If the station is switched to auto then the actuation signal is transmitted; if it is in manual mode a signal which has been entered manually (say, from keyboard) is transmitted. The signal from the bias station is connected to two units, a high signal selector and a low signal selector each of which has two inputs and one output. The signal from the low selector provides the set point for the DDC loop controlling the oil flow, the signal from the high selector provides the set point for the air flow controller (two cascade loops). A ratio unit is installed in the air flow measurement line.

DDC is not necessarily limited to simple feedback control as shown in Figure 2.6. It is possible to use techniques such as inferential, feed forward and adaptive or self-tuning control. Inferential control, illustrated in Figure 2.7, is the term applied to control where the variables on which the feedback control is to be based cannot be measured directly, but have to be 'inferred' from measurements of some other quantity.

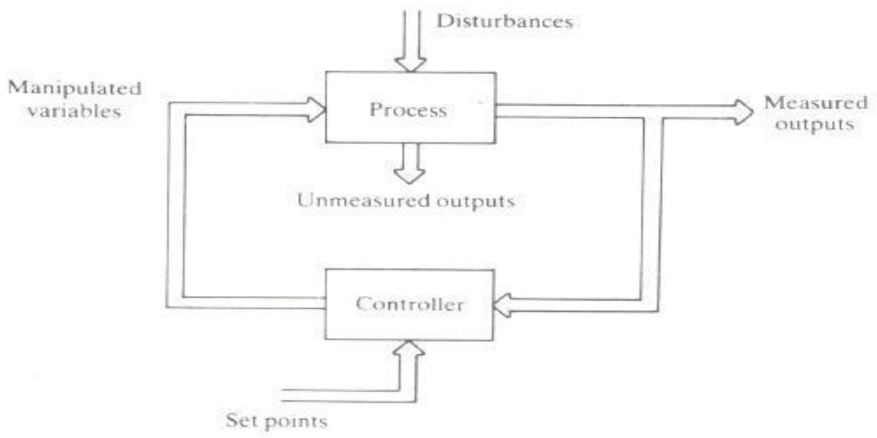


Figure 2.6: General structure of feedback control configuration.

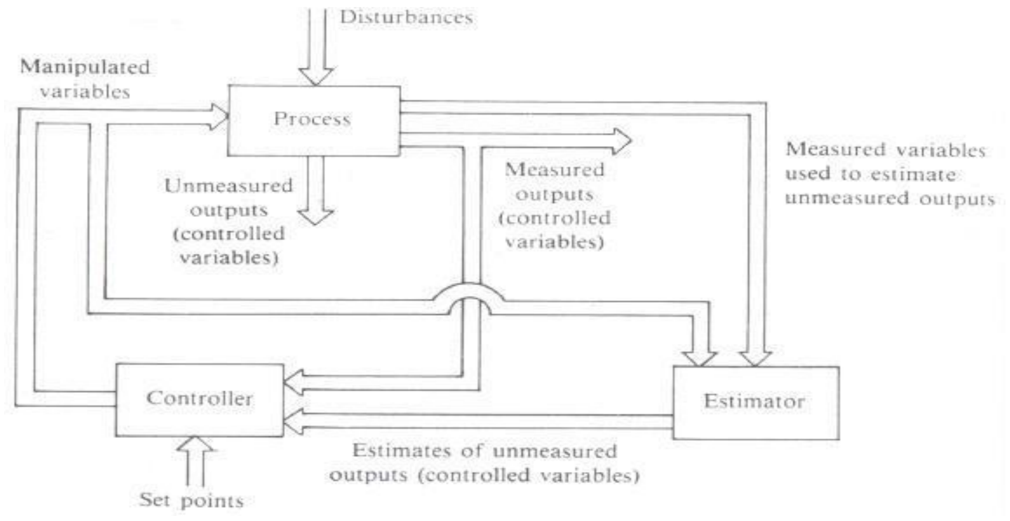

Disturbances
Manipulated variables
Process
Unmeasured outputs (controlled variables)
Measured outputs (controlled variables)
Measured variables used to estimate unmeasured outputs
Controller
Estimator
Estimates of unmeasured outputs (controlled variables)
Set points

Figure 2.7: General control of inferential control configurations.

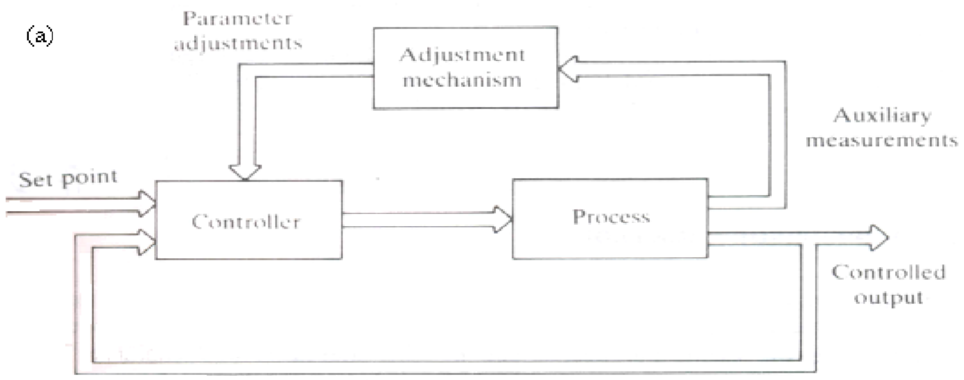
ADAPTIVE CONTROL:

Adaptive control can take several forms. Three of the most common are:

- Preprogrammed adaptive control (gain 5cheduled control);
- Self-tuning; and
- Model-reference adaptive control.

Programmed adaptive control is illustrated in Figure 2.10a. The adaptive, or adjustment, mechanism makes preset changes on the basis of changes in auxiliary process measurements. For example, in a reaction vessel a measurement of the level of liquid in the vessel (an indicator of the volume of liquid in the vessel) might be used to change the gain of the temperature controller; in many aircraft controls the measured air speed is used to select controller parameters according to a preset schedule.

An alternative form is shown in Figure 2.10b in which measurements of changes in the external environment are used to select the gain or other controller parameters. For example, in an aircraft auto stabilizer, control parameters may be changed according to the external air pressure.



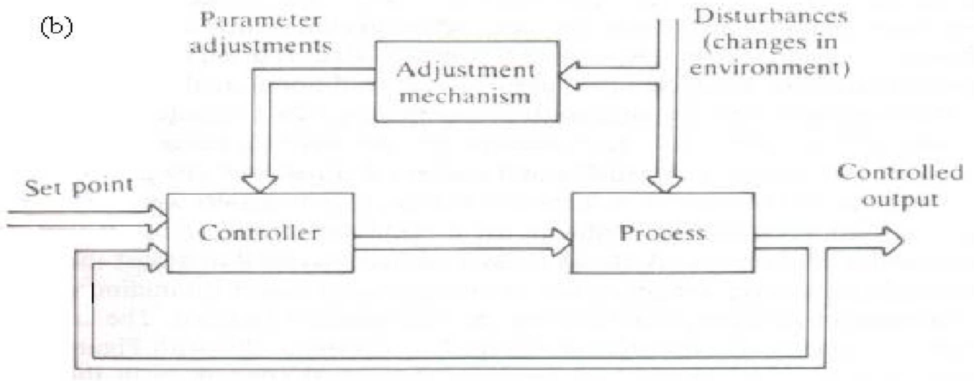


Figure2.10 Programmed adaptive control (gain scheduled):

(a) Auxiliary process measurements; (b) External environment (open loop).

Another example is the use of measurements of external temperature and wind velocities to adjust control parameters for a building environment control system. Adaptive control using self-tuning is illustrated in Figure 2.11 and uses identification techniques to achieve continual determination of the parameters of the process being controlled; changes in the process parameters are then used to adjust the actual controller. An alternative form of self-tuning is frequently found in commercial PID controllers (usually called auto tuning). The comparison may be based on a simple measure such as percentage overshoot or some more complex comparators. The model reference technique is illustrated in Figure 2.12; it relies on the ability to construct an accurate model of the process and to measure the disturbances which affect the process.

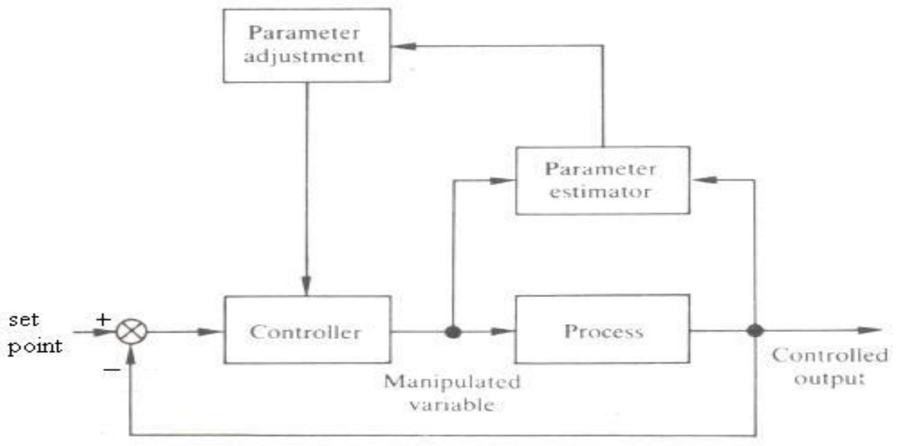


Figure 2.11: Self-tuning adaptive control.

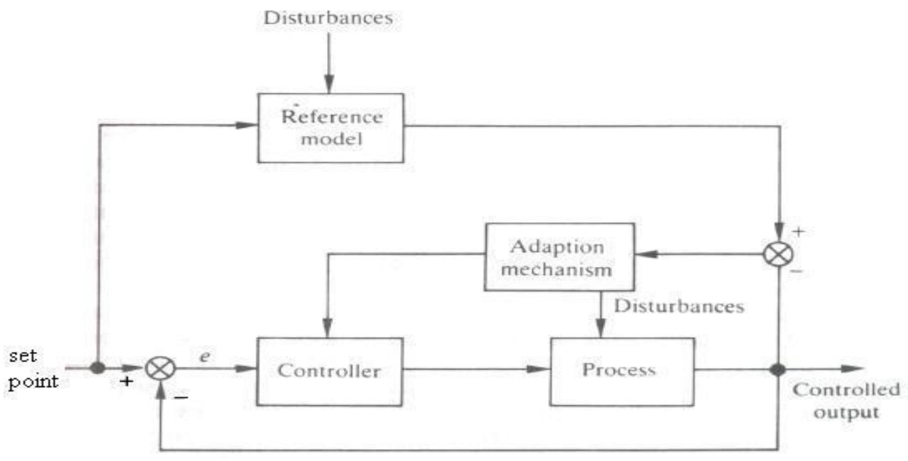


Figure 2.12: Model-reference adaptive control.

2.4 SUPERVISORY CONTROL:

The adoption of computers for process control has increased the range of activities that can be performed, for not only can the computer system directly control the operation of the plant, but also it can provide managers and engineers with a comprehensive picture of the status of the plant operations. It is in this supervisory role and in the presentation of information to the plant operator - large rooms full of dials and switches have been replaced by VDUs and keyboards - that the major changes have been made: the techniques used in the basic feedback control of the plant have changed little from the days when pneumatically operated three-term controllers were the norm. Direct digital control (DDC) is often simply the computer implementation of the techniques used for the traditional analog controllers.

Many of the early computer control schemes used the computer in a supervisory role and not for DDC. The main reasons for this were (a) computers in the early days were not always very reliable and caution dictated that the plant should still be able to run in the event of a computer failure; (b) computers were very expensive and it was not economically viable to use a computer to replace the analog control equipment in current use. A computer system that was used to adjust the set points of the existing analog control system in an optimum manner (to minimize energy or to maximize production) could perhaps be economically justified. The basic idea of supervisory control is illustrated in Figure 2.13 (compare this with Figure 2.4).

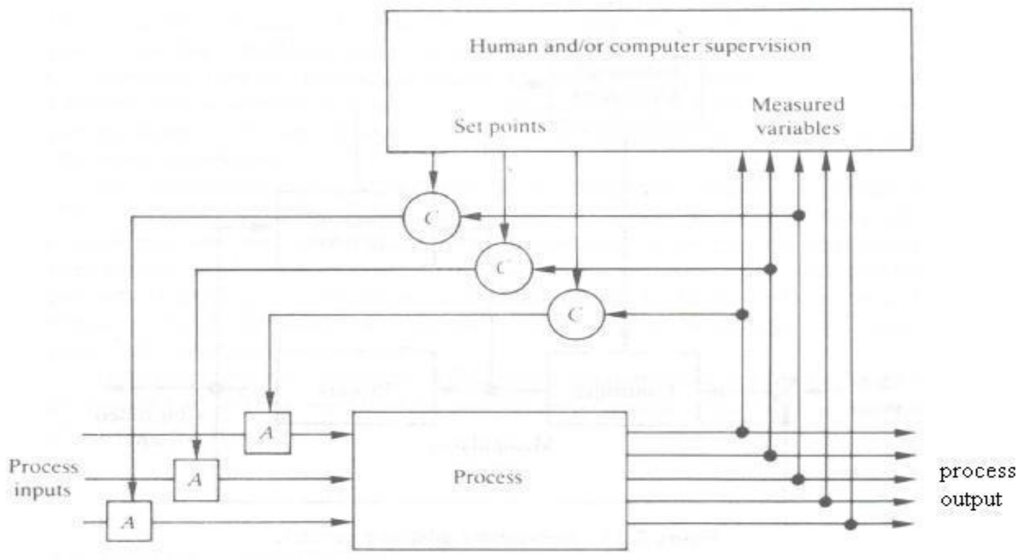


Figure 2.13: Supervisory control.

An example of supervisory control is shown in Figure 2.14. Two evaporators are connected in parallel and material in solution is fed to each unit. The purpose of the plant is to evaporate as much water as possible from the solution. Steam is supplied to a heat exchanger linked to the first evaporator and the steam for the second evaporator is supplied from the vapours boiled off from the first stage. To achieve maximum evaporation the pressures in the chambers must be as high as safety permits. However, it is necessary to achieve a balance between the two evaporators; if the first is driven at its maximum rate it may generate so much steam that the safety thresholds for the second evaporator are exceeded.

A supervisory control scheme can be designed to balance the operation of the two evaporators to obtain the best overall evaporation rate. Most applications of supervisory control are very simple and are based upon knowledge of the steady-state characteristics of the plant. In a few systems complex control algorithms have been used and have been shown to give increased plant profitability.

The techniques used have included optimization based on hill climbing, linear programming and simulations involving complex non-linear models of plant dynamics and economics.

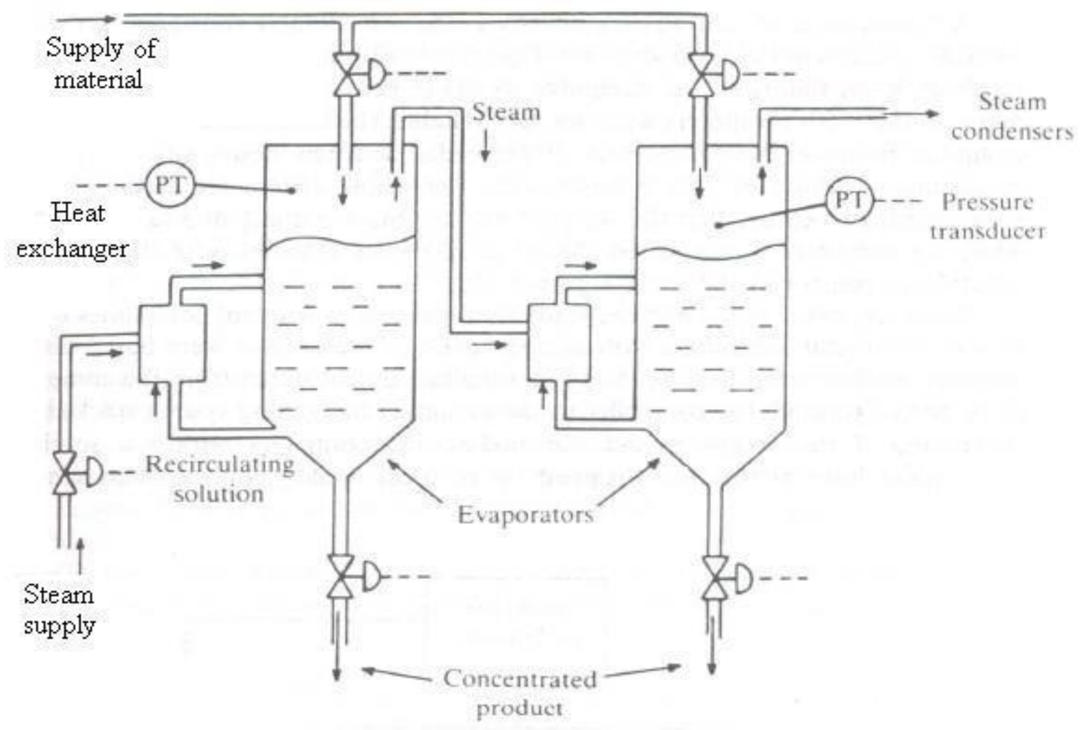


Figure 2.14: An evaporation plant.

2.5 CENTRALISED COMPUTER CONTROL:

Throughout most of the 1960s computer control implied the use of one central computer for the control of the whole plant. The reason for this was largely financial: computers were expensive. From the previous sections it should now be obvious that a typical computer-operation process involves the computer in performing many different types of operations and tasks. Although a general purpose computer can be programmed to perform all of the required tasks the differing time-scales and security requirements for the various categories of task make the programming job difficult, particularly with regard to the testing of software. For example, the feedback loops in a process may require calculations at intervals measured in seconds while some of the alarm and switching systems may require a response in less than 1 second; the supervisory control calculations may have to be repeated at intervals of several minutes or even hours; production management will want summaries at shift or daily intervals; and works management will require weekly or monthly analyses. Interrelating all the different time-scales can cause serious difficulties.

A consequence of centralized control was the considerable resistance to the use of DOC schemes in the form shown in Figure 2.4; with one central computer in the feedback loop, failure of the computer results in the loss of control of the 'whole plant. In the 1960s computers were not very reliable: the mean-time-to-failure of the computer hardware was frequently of the order of a few hours and to obtain a mean-time-to-failure of 3 to 6 months for the whole system required defensive programming to ensure that the system could continue running in a safe condition while the computer was repaired. Many of the early schemes were therefore for supervisory control as shown in Figure 2.13. However, in the mid 1960s the traditional process instrument companies began to produce digital controllers with analog back-up. These units were based on the standard analog controllers but allowed a digital control signal from the computer to be passed through the controller to the actuator: the analog system tracked the signal and if the computer did not update the controller within a specified (adjustable) interval the unit dropped on to local analog control. This scheme enabled DDC to be used with the confidence that if the computer failed, the plant could still be operated. The cost, however, was high in that two complete control systems had to be installed.

By 1970 the cost of computer hardware had reduced to such an extent that it became feasible to consider the use of dual computer systems (Figure 2.15). Here, in the event of failure of one of the computers, the other takes over. In some schemes the change-over is manual, in others automatic failure detection and change-over is incorporated. Many of these schemes are still in use. They do, however, have a number of weaknesses: cabling and interface equipment is not usually duplicated, neither is the software - in the sense of having independently designed and constructed programs - so that the lack of duplication becomes crucial. Automatic failure and change-over equipment when used becomes in itself a critical component. Furthermore, the problems of designing, programming, testing and maintaining the software are not reduced: if anything they are further complicated in that provision for monitoring ready for change-over has to be provided. The continued reduction of the cost of hardware and the development of the microprocessor has made multi-computer systems feasible. These fall into two types:

1. Hierarchical - Tasks are divided according to function, for example with one computer performing DDC calculations and being subservient to another which performs supervisory control.
2. Distributed - Many computers perform essentially similar tasks in parallel.

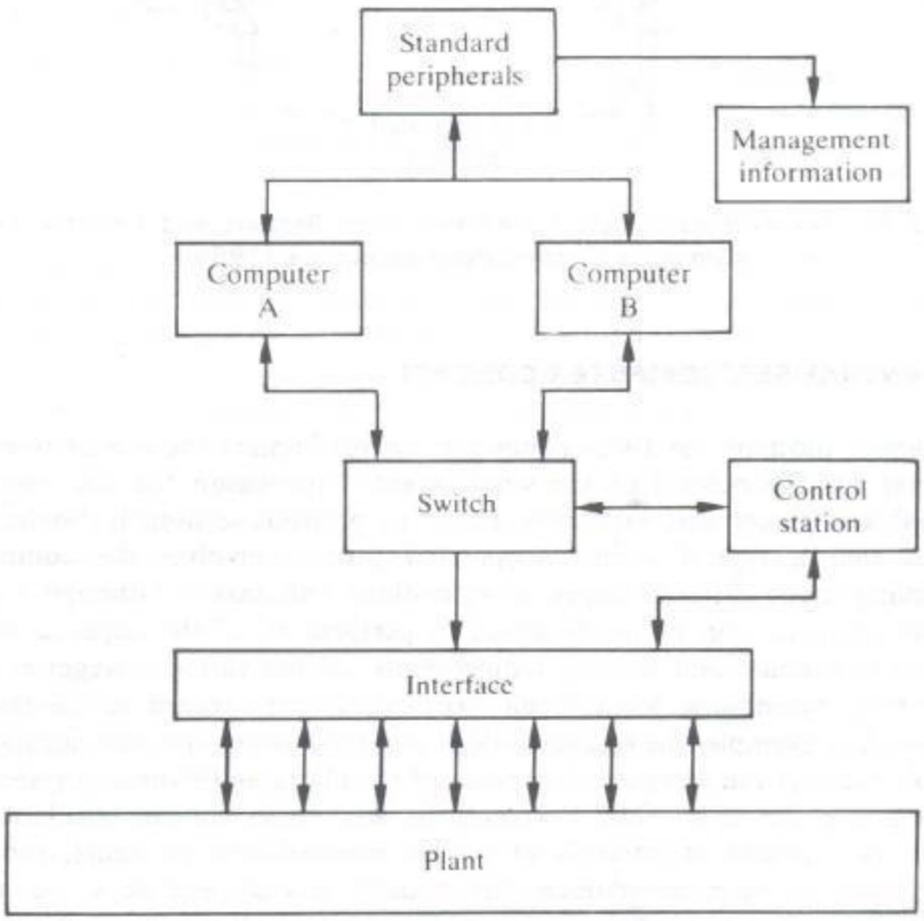


Figure 2.15: Dual computer scheme.

2.6 DISTRIBUTED SYSTEMS:

The underlying assumptions of the distributed approach are:

1. Each unit is carrying out essentially similar tasks to all the other units; and
2. In the event of failure or overloading of a particular unit all or some of the work can be transferred to other units.

In other words, the work is not divided by function and allocated to a particular computer as in hierarchical systems: instead, the total work is divided up and spread across several computers. This is a conceptually simple and attractive approach - many hands make light work - but it poses difficult hardware and software problems since, in order to gain the advantages of the approach, allocation of the tasks between computers has to be dynamic, that is there has to be some mechanism which can assess the work to be done and the present load on each computer in order to allocate work. Because each computer needs access to all the information in the system, high-bandwidth data highways are necessary. There has been considerable progress in developing such highways and the various types are discussed below:

Computer scientists and engineers are also carrying out considerable research on multi-processor computer systems and this work could lead to totally distributed systems becoming feasible. There is also a more practical approach to distributing the computing load whereby no attempt is made to provide for the dynamic allocation of resources but

instead a simple ad hoc division is adopted with, for example, one computer performing all non-plant input and output, one computer performing all DDC calculations, another performing data acquisition and yet another performing the control of the actuators. In most modern schemes a mixture of distributed and hierarchical approaches is used as shown in Figure 2.19. The tasks of measurement, DDC, operator communications, etc., are distributed among a number of computers which are linked together via a common serial communications highway and are configured in a hierarchical command structure. Five broad divisions of function are shown:

Level 1 all computations and plant interfacing associated with measurement and actuation. This level provides a measurement and actuation database for the whole system.

Level 2 All DDC calculations.

Level 3 all sequence calculations.

Level 4 Operator communications.

Level 5 Supervisory control

Level 6 Communications with other computer systems.

It is not necessary to preserve rigid boundaries; for example, a DDC unit may perform some sequencing or may interface directly to plant.

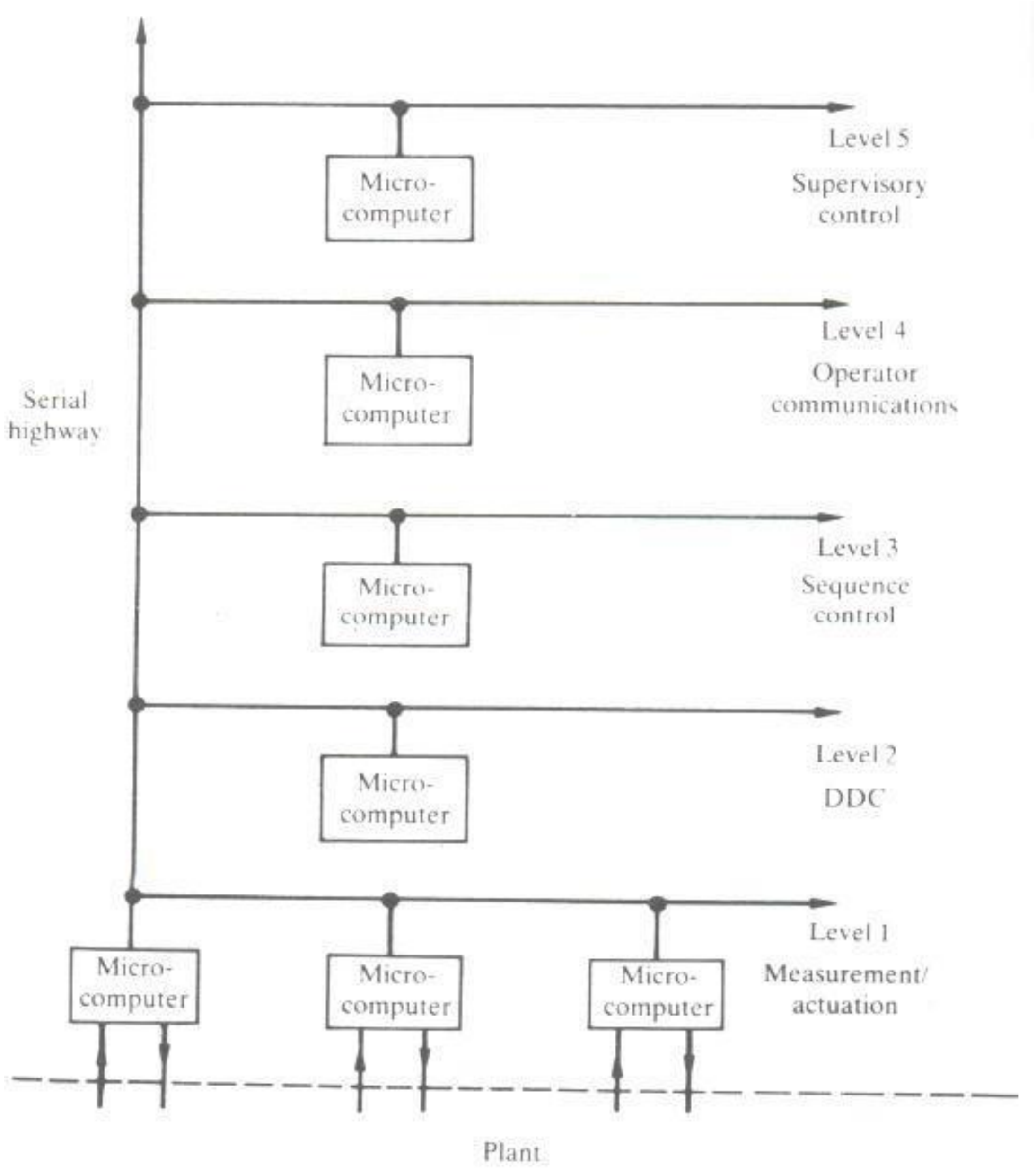


Figure 2.16: A distributed system.

The major advantages of this approach are:

1. The system capabilities are greatly enhanced by the sharing of tasks between processors - the burden of computation for a single processor becomes very great if all of the described control

features are included. One of the main computing loads is that of measurement scanning, filtering and scaling, not because anyone calculation is onerous but because of the large number of signals involved and the frequency at which the calculations have to be repeated. Separation of this aspect from the DDC, even if only into two processors, greatly enhances the number of control loops that can be handled. The DDC computer will collect measurements, already processed, via the communications link at a much lower frequency than that at which the measurement computer operates.

2. The system is much more flexible than the use of a single processor: if more loops are required or an extra operator station is needed, all that is necessary is to add more boxes to the communication link - of course the other units on the link will need to be updated to be aware of the additional items. It also allows standardization, since it is much easier to develop standard units for well-defined single tasks than for overall control schemes.

3. Failure of a unit will cause much less disruption in that only a small portion of the overall system will not be working. Provision of automatic or semiautomatic transfer to a back-up system is much easier.

4. It is much easier to make changes to the system, in the form of either hardware replacements or software changes. Changing large programs is hazardous because of the possibility of unforeseen side-effects: with the use of small modules such effects are less likely to occur and are more easily detected and corrected.

5. Linking by serial highway means that the computer units can be widely dispersed: hence it is unnecessary to bring cables carrying transducer signals to a central control room.

2.7 HUMAN –COMPUTER INTERFACE:

The key to the successful adoption of a computer control scheme is often the facilities provided for the plant operator or user of the system. A simple and clear system for the day-to-day operation of the plant must be provided. All the information relevant to the current state of its operation should be readily available and facilities to enable interaction with the plant - to change set points, to adjust actuators by hand, to acknowledge alarm conditions, etc. - should be provided. A large proportion of the design and programming effort goes into the design and construction of operator facilities and the major process control equipment companies have developed extensive schemes for the presentation of information.

A typical operator station has specially designed keyboards and several display and printer units; extensive use is made of color displays and mimic diagrams; video units are frequently provided to enable the operator to see parts of the plant (Jovic, 1986). The standard software packages typically provide a range of display types: an alarm overview presenting information on the alarm status of large areas of the plant; a number of area displays presenting information on the control systems associated with each area; and loop displays giving extensive information on the details of a particular control loop. The exact nature of the displays is usually determined by the engineer responsible for the plant or part of the plant.

The plant manager requires access to different information: hard copy printouts - including graphs - that summarize the day-to-day operation of the plant and also provide a permanent plant operating history. Data presented to the manager will frequently have been analyzed statistically to provide more concise information and

to make decision-making more straightforward. The manager will be interested in assessing the economic performance of the plant and in determining possible improvements in plant operation. The design of user interfaces is a specialist area. The safe operation of complex systems such as aircraft, nuclear power stations, chemical plants, air traffic control systems and other traffic control systems can be crucially affected by the way in which information is presented to the operator.

2.8 BENEFITS OF COMPUTER CONTROL SYSTEMS:

Before the widespread availability of microprocessors, computer control was expensive and a very strong case was needed to justify the use of computer control rather than conventional instrumentation. In some cases computers were used because otherwise plant could not have been made to work profitably: this is particularly the case with large industrial processes that require complex sequencing operations. The use of a computer permits the repeatability that is essential, for example, in plants used for the manufacture of drugs. In many applications flexibility is important - it is difficult with conventional systems to modify the sequencing procedure to provide for the manufacture of a different product.

Flexibility is particularly important when the product or the product specification may have to be changed frequently: with a computer system it is simple to maintain a database containing the product recipes and thus to change to a new recipe quickly and reliably.

The application of computer control systems to many large plants has frequently been justified on the grounds that even a small increase in productivity (say I or 2070) will more than pay for the computer system. After installation it has frequently been difficult to establish that an improvement has been achieved; sometimes production has decreased, but the computer proponents have then argued that but for the introduction of the computer system production would have decreased by a greater amount! Some of the major benefits to accrue from the introduction of computer systems have been in the increased understanding of the behavior of the process that has resulted from the studies necessary to design the computer system and from the information gathered during running. This has enabled supervisory systems to keep the plant running at an operating point closer to the desired point to be designed.

The other main area of benefit has been in the control of the starting and stopping of batch operations in that computer-based systems have generally significantly reduced the dead time associated with batch operations. The economics of computer control have been changed drastically by the microprocessor in that the reduction in cost and the improvement in reliability have meant that computer-based systems are the first choice in many applications. Indeed, microprocessor-based instrumentation is frequently cheaper than the equivalent analog unit. The major costs of computer control are now no longer the computer hardware, but the system design and the cost of software: as a consequence attention is shifting towards greater standardization of design and of software products and the development of improved techniques for design (particularly software design) and for software construction and testing. The availability of powerful, cheap and highly reliable computer hardware and communications systems makes it possible to conceive and construct large, complex, computer-based control systems. The complexity of such systems raises concern about their dependability and safety.

Recommended Question:

1. List the advantages and disadvantages of using DDC.
2. 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. 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?

4. 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?
5. List the characteristics of Batch process and continuous process.

MODULE- 2

Computer Hardware Requirements for RTS

Introduction, General Purpose Computer, Single Chip Microcontroller, Specialized Processors, Process –Related Interfaces, Data Transfer Techniques, Communications, Standard Interface.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.

3.1 COMPUTER HARDWARE REQUIREMENTS FOR RTS.

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 specialized digital processor. Specialized 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 specialized 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.

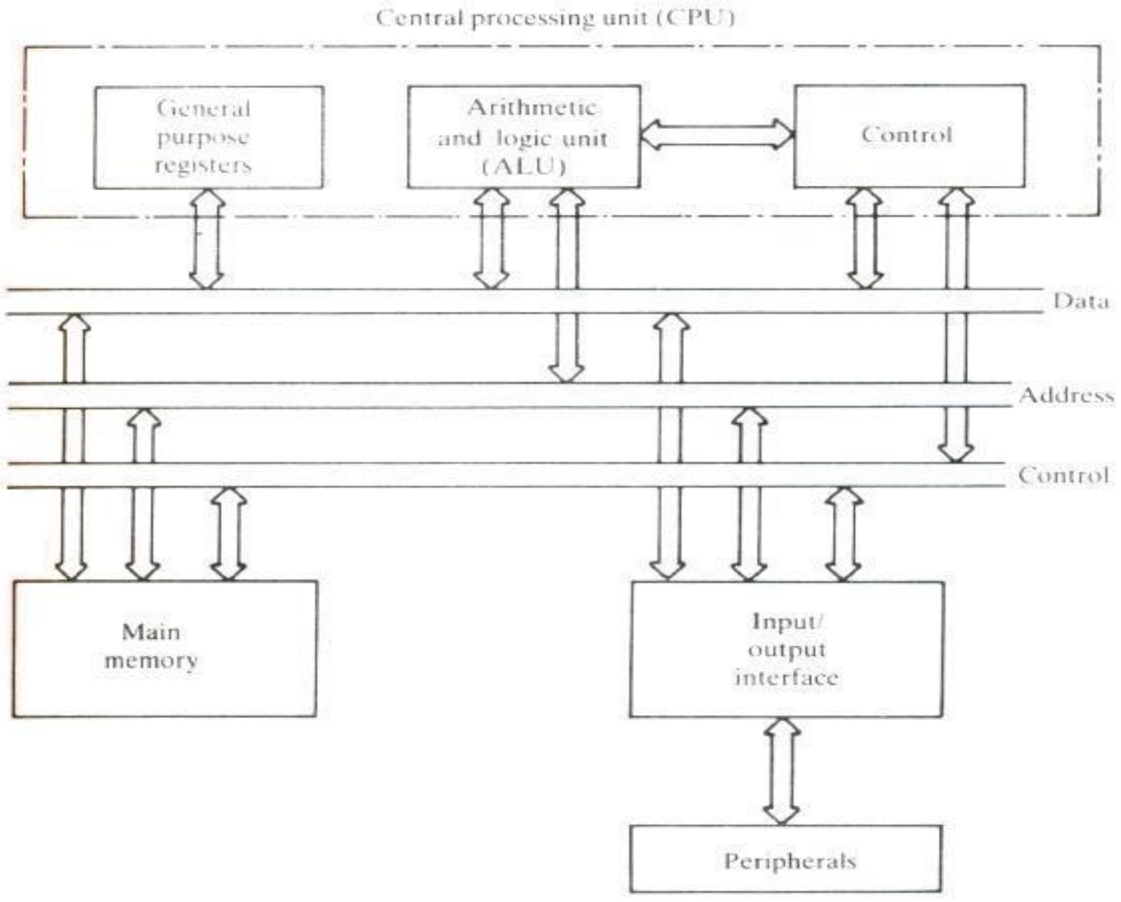

Central processing unit (CPU)
General purpose registers
Arithmetic and logic unit (ALU)
Control
Data
Address
Control
Main memory
Input/ output interface
Peripherals

Figure: 3.1 Schematic diagram of a general purpose digital computer.

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 word length 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.

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. 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.

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.

BUS STRUCTURE:

Buses are characterized into 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.

3.3 SINGLE CHIP MICROCONTROLLER:

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.

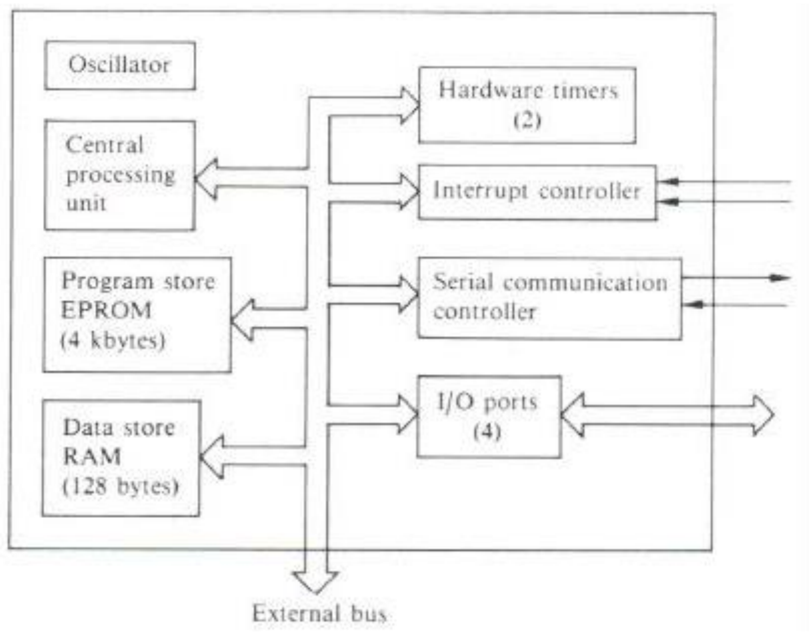


Figure 3.2: A typical single-chip computer.

3.4 SPECIALIZED PROCESSORS:

Specialized 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 veri fled).
- No interrupts - all event handling is done using polling (again interrupts make formal verification impossible).
- No dynamic memory allocation.

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.

3.5 PARELLEL COMPUTERS:

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

Single instruction stream, multiple data stream.

Multiple instruction stream, single data stream.

Multiple instruction stream, multiple data stream.

These are illustrated in Figure 3.3 where the traditional architecture characterized 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.

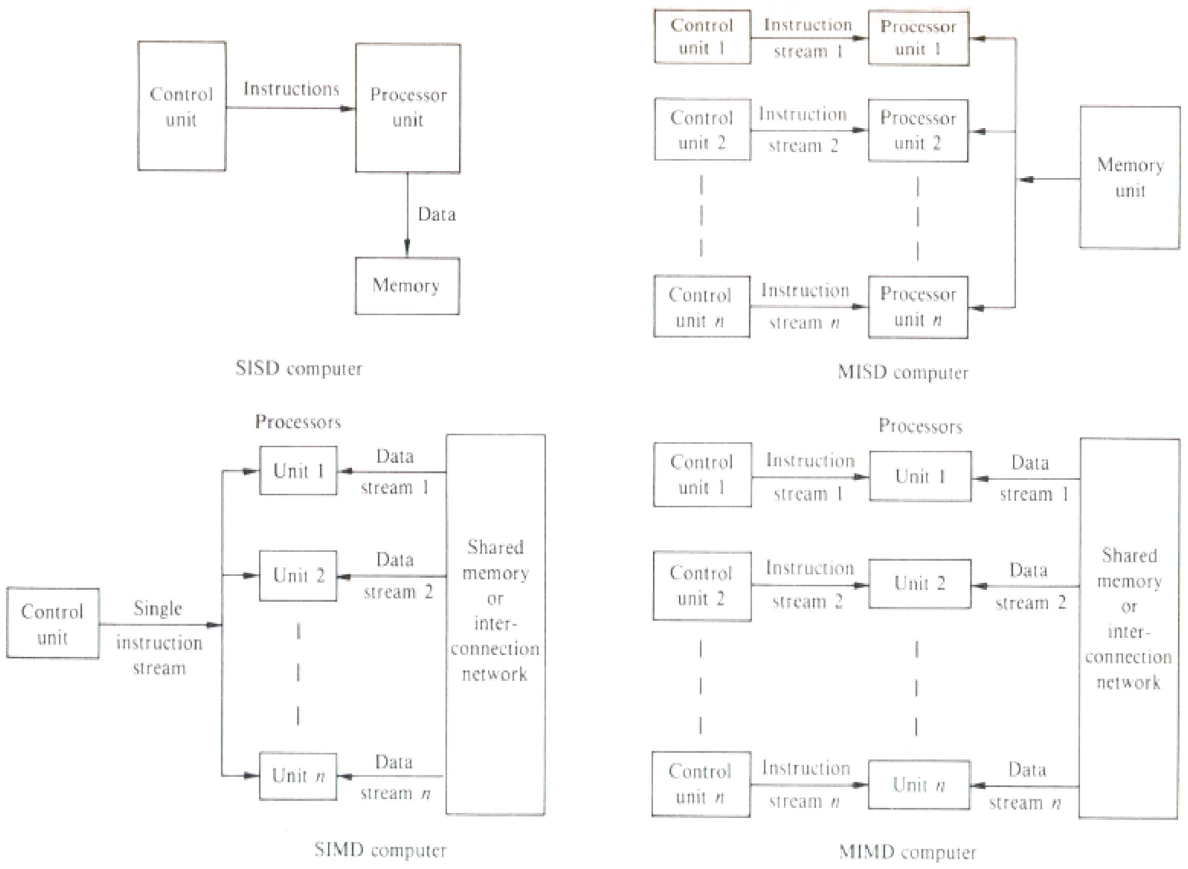


Figure 3.3: Computer system architecture.

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 synchronized 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.

3.6 DIGITIAL 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 optimized 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.

3.7 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 generalized 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 mill volts; 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 or

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 net work: it is, however, characterized 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.

3.8 DIGITIAL 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 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.

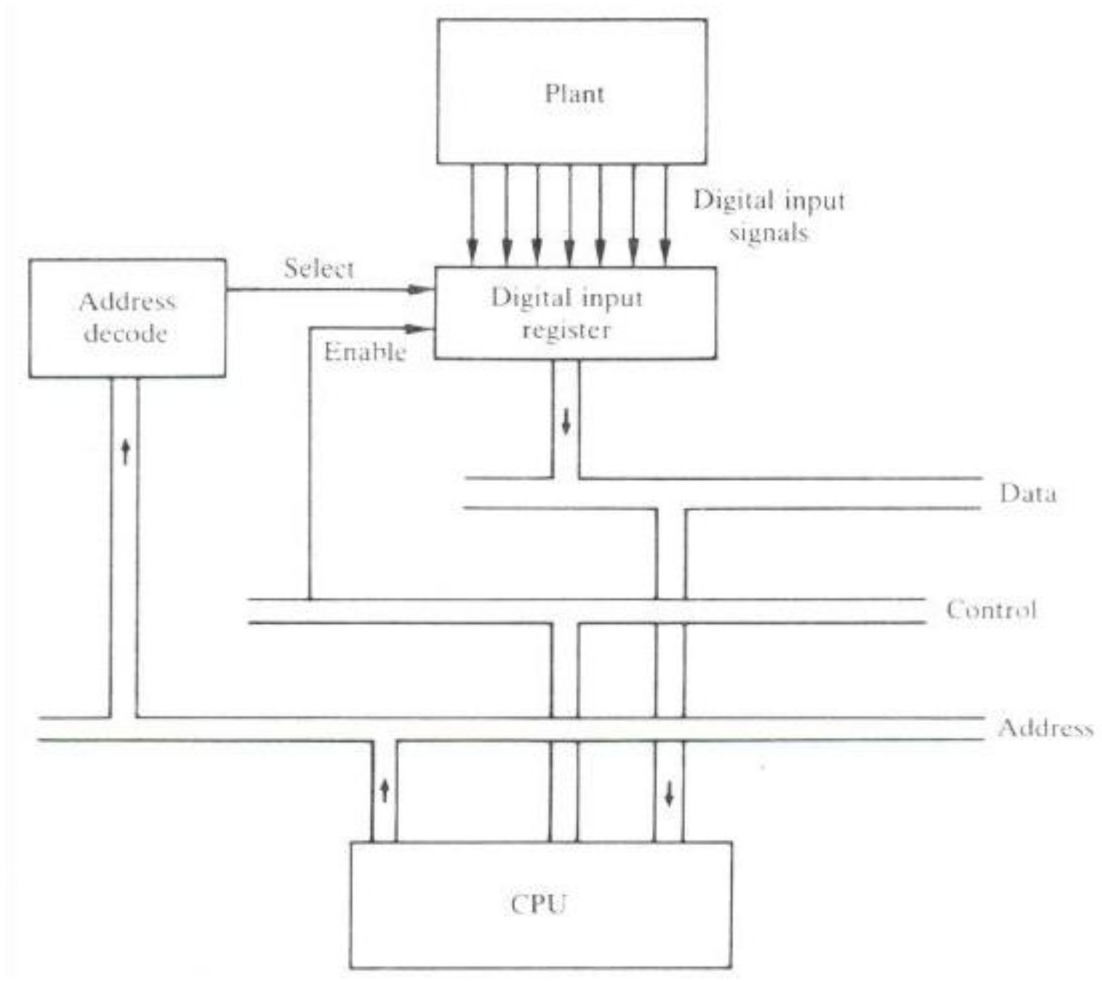


Figure 3.4: A simple digital input interface.

Figure 3.4 shows a system that provides information only on demand from the computer: it cannot indicate 10 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 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.

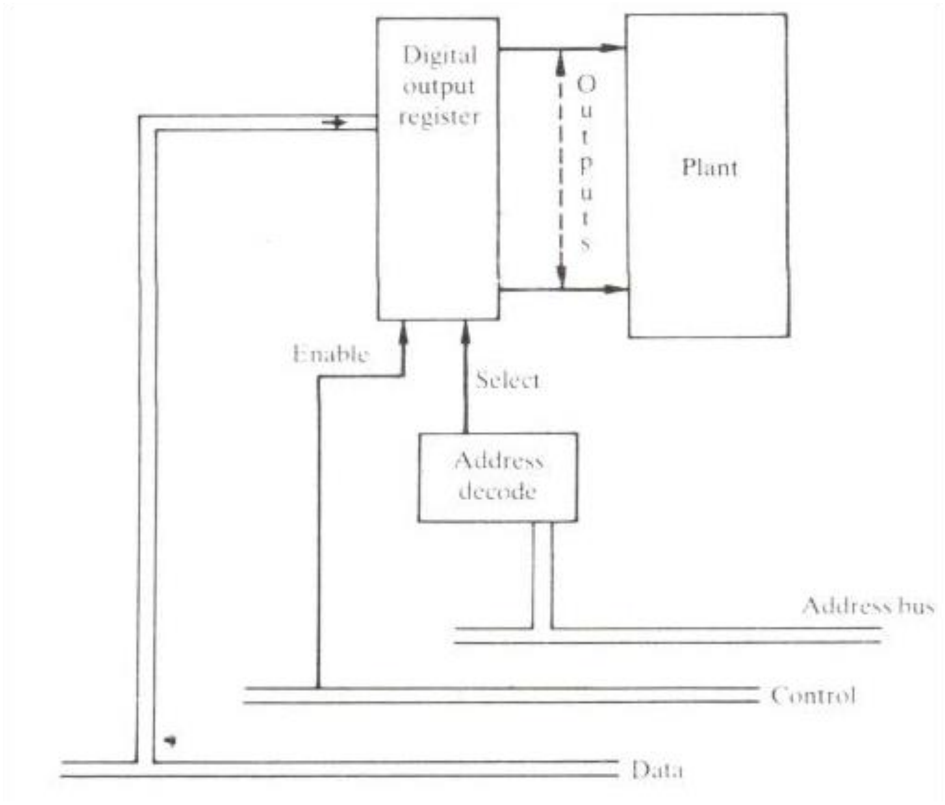


Figure 3.6: A simple digital output interface.

3.9 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 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.

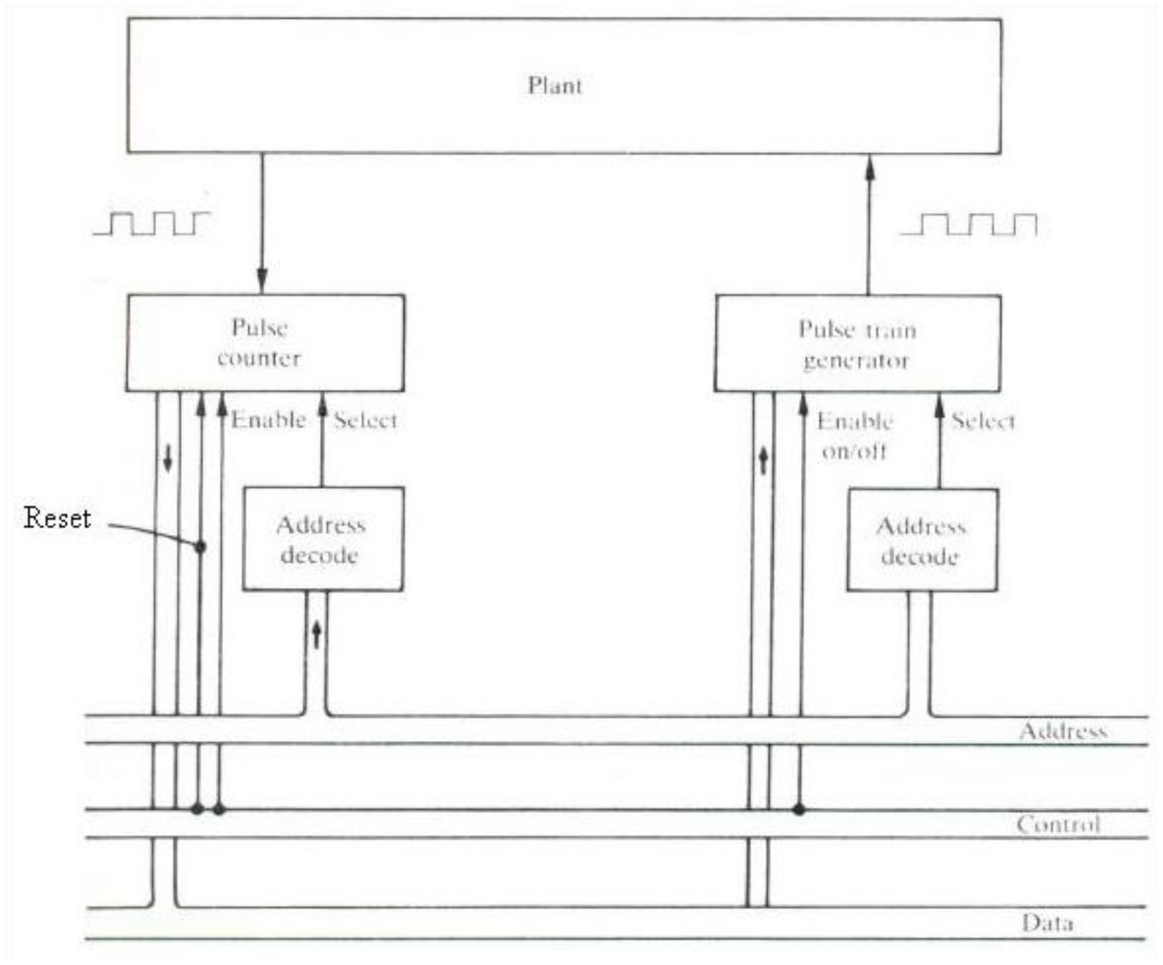


Figure 3.7: Pulse input and output interfaces.

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.

3.10 ANALOG INTERFACES:

The conversion of analog measurements to digital measurements involves two operations: sampling and quantization. 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 (I microsecond), and in response the ADC switches the 'sample-hold' into SAMPLE for a short period after which the quantization process commences. Quantization 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.

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.

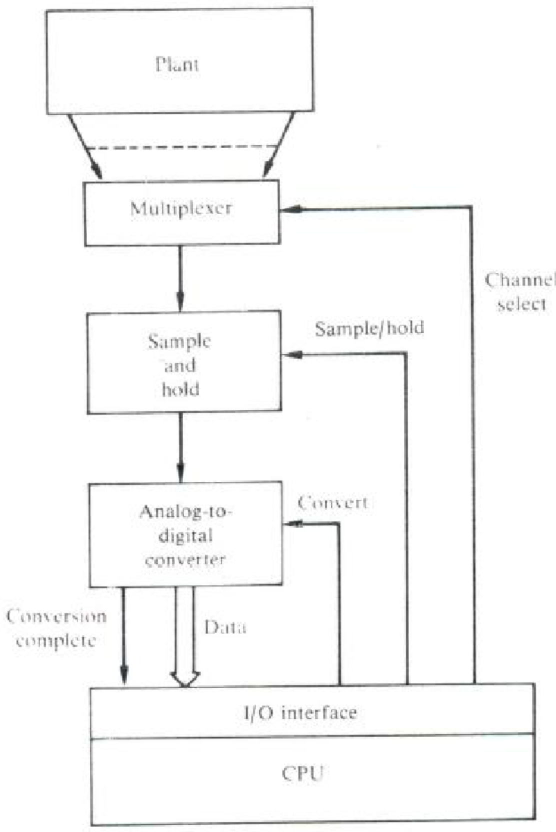


Figure 3.8: Analog input system.

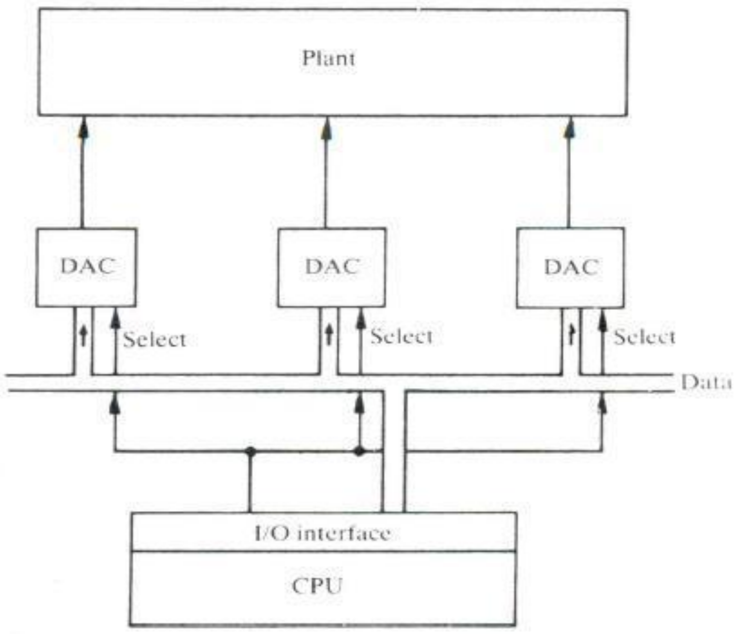


Figure 3.9: Analog output system.

3.11 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' mayor 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.

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.

3.12 DATA TRANSFER TECHNIQUES:

Although the meaning of the data transmitted by the various processes, the operator and computer peripherals differ, 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 synchronies 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.

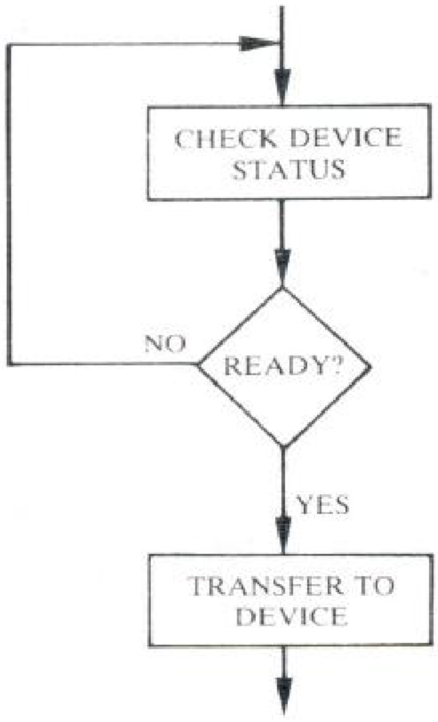


Figure 3.10: Conditional transfer (busy wait).

3.13 COMMUNCIATIONS:

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 land 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).

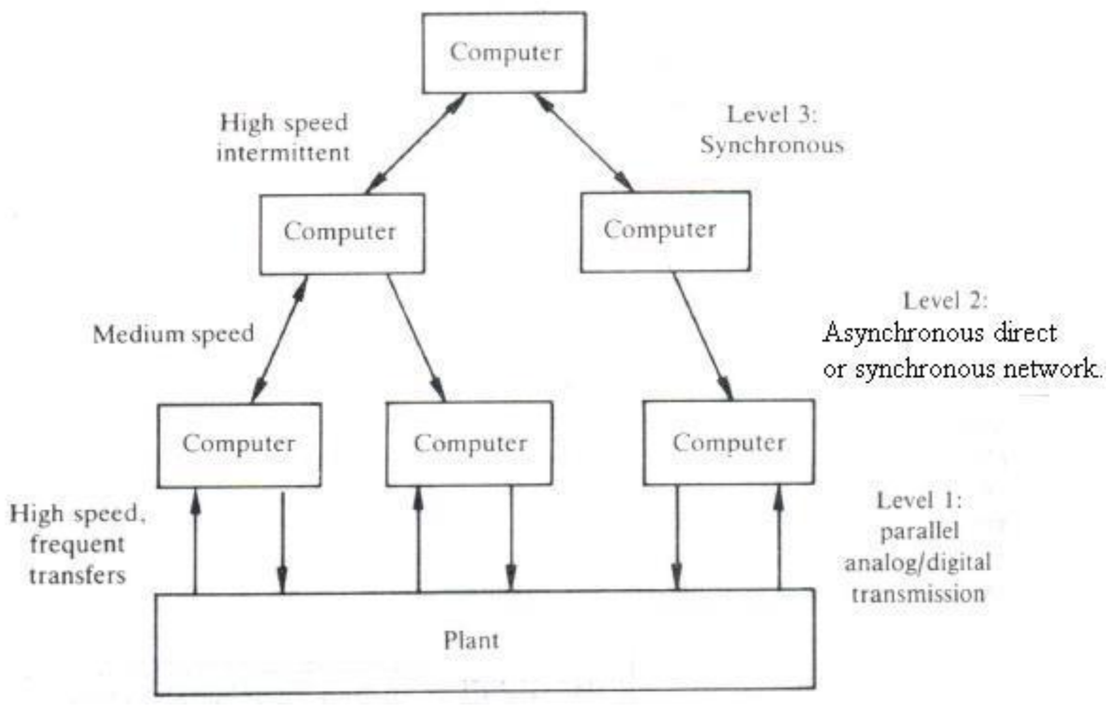

Computer
High speed intermittent
Level 3: Synchronous
Computer
Computer
Medium speed
Level 2: Asynchronous direct or synchronous network.
Computer
Computer
Computer
High speed, frequent transfers
Level 1: parallel analog/digital transmission
Plant

Figure 3.20: Data transmission links.

Serial communications can be characterized 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

3.14 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-ll 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 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 standardized as the IEEE 488 bus system.

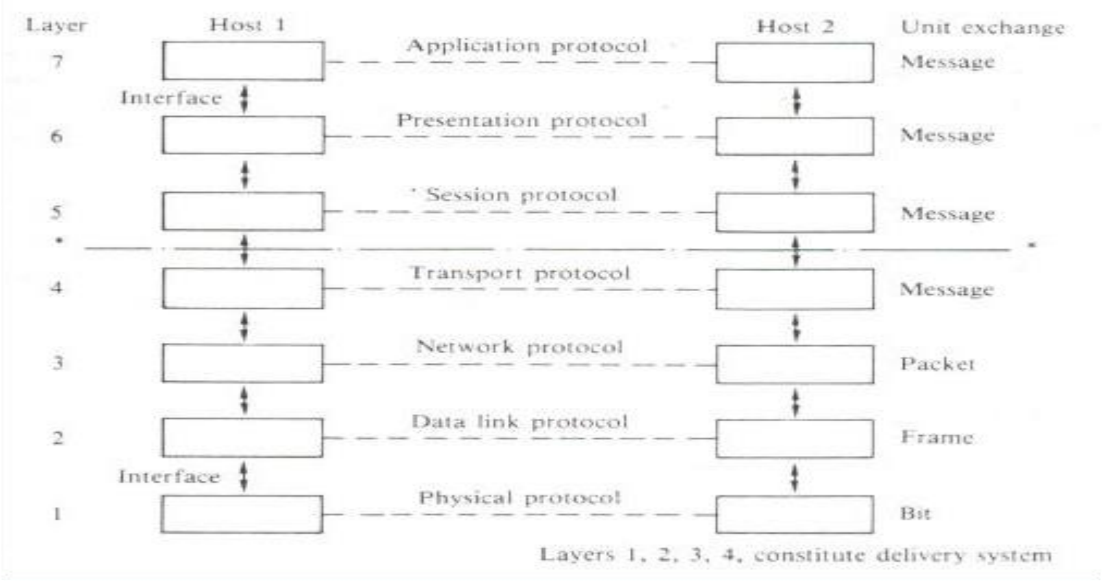
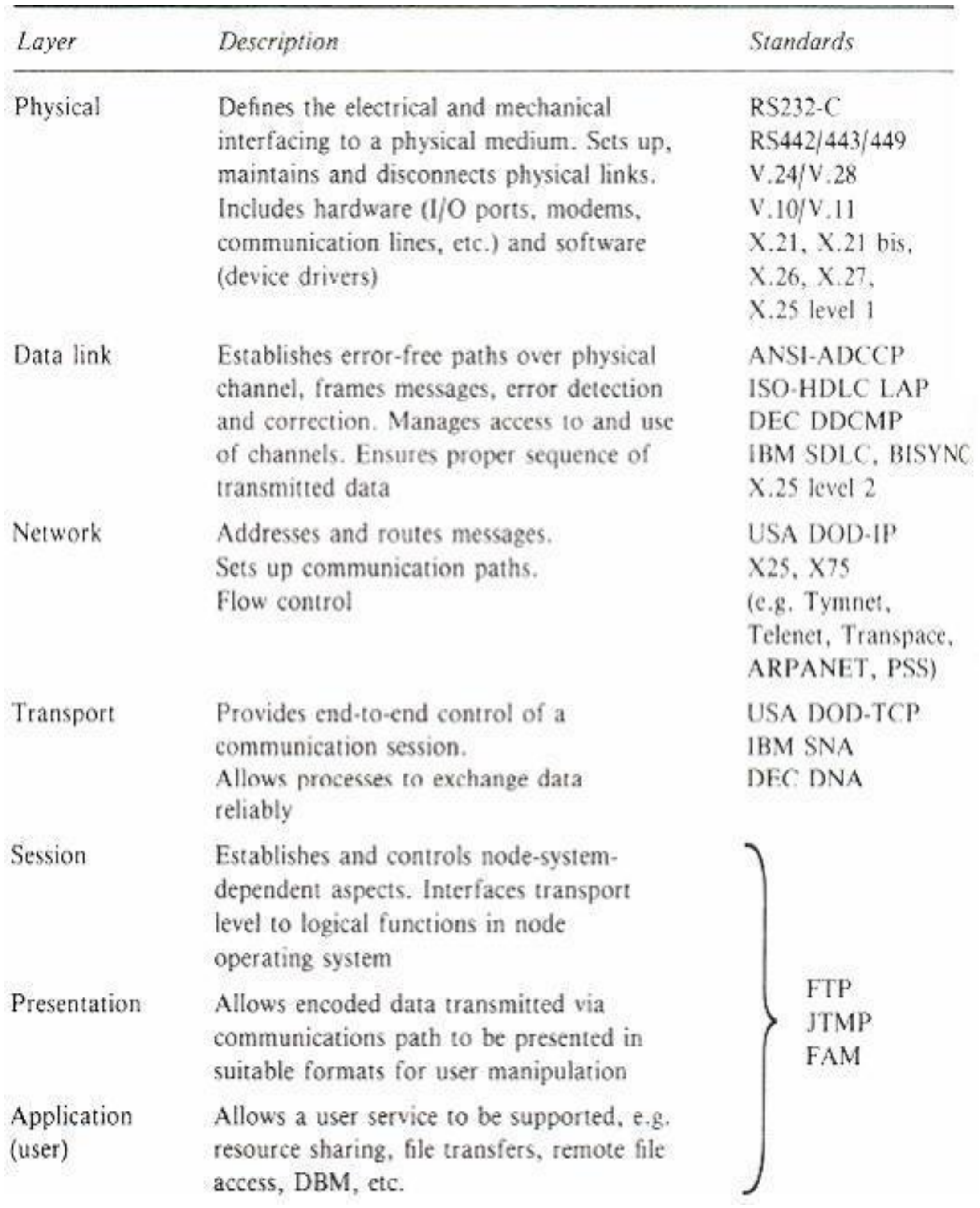


Figure 3.25: ISO seven - layer model.



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.	

Table: 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 Organization for Standardization) 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.

Recommended questions:

1. 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)).
2. 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. Twenty analog signals from a plant have to be processed (sampled and digitized) every 1s. 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.
4. 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?
5. Why is memory protection important in real-time systems?
6. What methods can be used to provide memory protection?

MODULE- 3

Languages For Real –Time Applications

Introduction, Syntax Layout and Readability, Declaration and Initialization of Variables and Constants, Modularity and Variables, Compilation , Data Type, Control Structure, Exception Handling, Low –Level Facilities, Co routines, Interrupts and Device Handling, Concurrency, Real –Time Support, Overview of Real –Time Languages.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.
3. **Real time Systems Development**, Rob Williams, Elsevier, 2006.

LANGUAGES FOR REAL-TIME APPLICATIONS

4.1 INTRODUCTION

Languages are an important implementation tool for all systems that include embedded computers. To understand fully methods for designing software for such systems one needs to have a sound understanding of the range of implementation languages available and of the facilities which they offer. The range of languages with features for real-time use continues to grow, as do the range and type of features offered. In this chapter we concentrate on the fundamental requirements of a good language for real-time applications and will illustrate these with examples drawn largely from Modula-2 and Ada. Producing safe real-time software places heavy demands on programming languages. Real-time software must be reliable: the failure of a real-time system can be expensive both in terms of lost production, or in some cases, in the loss of human life (for example, through the failure of an aircraft control system).

Real-time systems are frequently large and complex, factors which make development and maintenance costly. Such systems have to respond to external events with a guaranteed response time; they also involve a wide range of interface devices, including non-standard devices. In many applications efficiency in the use of the computer hardware is vital in order to obtain the necessary speed of operation. Early real-time systems were necessarily programmed using assembly level languages, largely because of the need for efficient use of the CPU, and access interface devices and support interrupts. Assembly coding is still widely used for small systems with very high computing speed requirements, or for small systems which will be used in large numbers. In the latter case the high cost of development is offset by the reduction in unit cost through having a small, efficient, program. Dissatisfaction with assemblers (and with high-level languages such as FORTRAN which began to be used as it was recognized that for many applications the advantages of high-level languages outweighed their disadvantages) led to the development of new languages for programming embedded computers.

The limitation of all of them is that they are designed essentially for producing sequential programs and hence rely on operating system support for concurrency. The features that a programmer demands of a real-time language subsume those demanded of a general purpose language and so many of the features described below are also present (or desirable) in languages which do not support real-time operations. Barnes (1976) and Young (1982) divided the requirements that a user looked for in a programming language into six general areas. These are listed below in order of importance for real-time applications:

- Security.
- Readability.
- Flexibility.
- Simplicity.
- Portability.
- Efficiency.

In the following sections we will examine how the basic features of languages meet the requirements of the user as given above. The basic language features examined are:

- Variables and constants: declarations, initialization.
- Data types - including structured types and pointers.
- Control structures and program layout and syntax.
- Scope and visibility rules.
- Modularity and compilation methods.

- Exception handling.

A language for real-time use must support the construction of programs that exhibit concurrency and this requires support for:

- Construction of modules (software components).
- Creation and management of tasks.
- Handling of interrupts and devices.
- Intertask communication.
- Mutual exclusion.
- Exception handling.

4.1.1 SECURITY:

Security of a language is measured in terms of how effective the compiler and the run-time support system is in detecting programming errors automatically. Obviously there are some errors which cannot be detected by the compiler regardless of any features provided by the language: for example, errors in the logical design of the program. The chance of such errors occurring is reduced if the language encourages the programmer to write clear, well-structured, code. Language features that assist in the detection of errors by the compiler include:

- good modularity support;
- enforced declaration of variables;
- good range of data types, including sub-range types;
- typing of variables; and
- unambiguous syntax.

It is not possible to test software exhaustively and yet a fundamental requirement of real-time systems is that they operate reliably. The intrinsic security of a language is therefore of major importance for the production of reliable programs. In real-time system development the compilation is often performed on a different computer than the one used in the actual system, whereas run-time testing has to be done on the actual hardware and, in the later stages, on the hardware connected to plant. Run-time testing is therefore expensive and can interfere with the hardware development program. Economically it is important to detect errors at the compilation stage rather than at run-time since the earlier the error is detected the less it costs to correct it. Also checks done at compilation time have no run-time overheads.

4.1.2 READABILITY:

Readability is a measure of the ease with which the operation of a program can be understood without resort to supplementary documentation such as flowcharts or natural language descriptions. The emphasis is on ease of reading because a particular segment of code will be written only once but will be read many times. The benefits of good readability are:

- Reduction in documentation costs: the code itself provides the bulk of the documentation. This is particularly valuable in projects with a long life expectancy in which inevitably there will be a series of modifications. Obtaining up-to-date documentation and keeping documentation up to date can be very difficult and costly.
- Easy error detection: clear readable code makes errors, for example logical errors, easier to detect and hence increases reliability.
- Easy maintenance: it is frequently the case that when modifications to a program are required the person responsible for making the modifications was not involved in the original design - changes can only be made quickly and safely if the operation of the program is clear.

4.1.3 FLEXIBILITY:

A language must provide all the features necessary for the expression of all the operations required by the application without requiring the use of complicated constructions and tricks, or resort to assembly level code inserts. The flexibility of a language is a measure of this facility. It is particularly important in real-time systems since frequently non-standard I/O devices will have to be controlled. The achievement of high flexibility can conflict with achieving high security. The compromise that is reached in modern languages is to provide high flexibility and, through the *module* or *package* concept, a means by which the low-level (that is, insecure) operations can be hidden in a limited number of self-contained sections of the program.

4.1.4 SIMPLICITY:

In language design, as in other areas of design, the simple is to be preferred to the complex. Simplicity contributes to security. It reduces the cost of training, it reduces the probability of programming errors arising from misinterpretation of the language features, it reduces compiler size and it leads to more efficient object code. Associated with simplicity is consistency: a good language should not impose arbitrary restrictions (or relaxations) on the use of any feature of the language.

4.1.5 PORTABLITILY:

Portability, while desirable as a means of speeding up development, reducing costs and increasing security, is difficult to achieve in practice. Surface portability has improved with the standardization agreements on many languages. It is often possible to transfer a program in source code form from one computer to another and find that it will compile and run on the computer to which it has been transferred. There are, however, still problems when the word lengths of the two machines differ and there may also be problems with the precision with which numbers are represented even on computers with the same word length.

Portability is more difficult for real-time systems as they often make use of specific features of the computer hardware and the operating system. A practical solution is to accept that a real-time system will not be directly portable, and to Restrict the areas of non-portability to specific modules by restricting the use of low level features to a restricted range of modules. Portability can be further enhanced by writing the application software to run on a virtual machine, rather than for a specific operating system.

4.1.6 EFFICIENCY:

In real-time systems, which must provide a guaranteed performance and meet specific time constraints, efficiency is obviously important. In the early computer control systems great emphasis was placed on the efficiency of the coding - both in terms of the size of the object code and in the speed of operation - as computers were both expensive and, by today's standards, very slow. As a consequence programming was carried out using assembly languages and frequently 'tricks' were used to keep the code small and fast. The requirement for generating efficient object code was carried over into the designs of the early real-time languages and in these languages the emphasis was on efficiency rather than security and readability. The falling costs of hardware and the increase in the computational speed of computers have changed the emphasis. Also in a large number of real-time applications the concept of an efficient language has changed to include considerations of the security and the costs of writing and maintaining the program; speed and compactness of the object code have become, for the majority of applications, of secondary importance.

4.1.7 SYNTAX LAYOUT AND READAILITY:

The language syntax and its layout rules have a major impact on the readability of code written in the language. Consider the program fragment given below: BEGIN

```
NST: = TICKS ( ),. ST;
T: =TICKS ()+ST;
LOOP
WHILE TICKS ( )< NST DO (* nothing *) END;
T: =TICKS ();
C C;
NST: = T+ST;
IF KEYPRESSED ( ) THEN EXIT;
END;
END;
END;
```

Without some explanation and comment the meaning is completely obscure. By using long identifiers instead of, for example N S T and ST, it is possible to make the code more readable.

```
BEGIN
NEXTSAMPLETIME: = TICKSO+SAMPLETIME;
TIME: =TICKS () +SAMPLETIME;
LOOP
WHILE TICKSO< NEXTSAMPLETIME DO (* NOTHING
*) END;
```

```
TIME: =TICKSO;
CONTROLCALCULATION;
NEXTSAMPLETIME: =TIME+SAMPLETIME;
IF KEYPRESSEDOTHEN EXIT;
END;
END;
END;
```

The meaning is now a little clearer, although the code is not easy to read because it is entirely in upper case letters. We find it much easier to read lower case text than upper case and hence readability is improved if the language permits the use of lower case text. It also helps if we can use a different case (or some form of distinguishing mark) to identify the reserved words of the language. Reserved words are those used to identify
particular language constructs, for example repetition statements, variable declarations, etc. In the next version we use upper case for the reserved words and a mixture of upper and lower case for user-defined entities.

```
BEGIN
NextSampleTime: = Ticks ( ) +Sample Time;
Time: =Ticks ( ) +Sample Time;
LOOP
WHILE Ticks ( ) < NextSampleTime DO (* nothing *)
END;
Time: =Ticks ( );
Control Calculation;
NextSampleTime: = Time + Sample Time;
IF Key Pressed ( ) THEN EXIT;
END;
END;
END;
```

The program is now much easier to read in that we can easily and quickly pick out the reserved words. It can be made even easier to read if the language allows embedded spaces and tab characters to be used to improve the layout.

4.2 DECLARATION AND INTIALIZATION OF VARIABLES AND CONSTANTS.

DECLARATION:

The purpose of declaring an entity used in a program is to provide the compiler with information on the storage requirements and to inform the system explicitly of the names being used. Languages such as Pascal, Modula-2 and Ada require all objects to be specifically declared and a type to be associated with the entity when it is declared. The provision of type information allows the compiler to check that the entity is used only in operations associated with that type. If, for example, an entity is declared as being of type REA L and then it is used as an operand in logical operation, the compiler should detect the type incompatibility and flag the statement as being incorrect. Some older languages, for example BASIC and FORTRAN, do not require explicit declarations; the first use of a name is deemed to be its declaration. In FORTRAN explicit declaration is optional and entities can be associated with a type jf declared. If entities are not declared then implicit typing takes place: names beginning with the letters I-N are assumed to be integer numbers; names beginning with any other letter are assumed to be real numbers.

Optional declarations are dangerous because they can lead to the construction of syntactically correct but functionally erroneous programs. Consider the following program fragment:

```
100 ERROR=0
   .......
200 IF X=Y THEN GOTO 300
250 EROR=1
300...
```

In FORTRAN (or BASIC), ERROR and EROR will be considered as two different variables whereas the programmer's intention was that they should be the same – the variable ER 0 R in line 250 has been mistyped. FORTRAN compilers cannot detect this type of error and it is a *characteristic* error of FORTRAN. Many organizations which use FORTRAN extensively avoid such errors by insisting that all entities are declared and the code is processed by a preprocessor which checks that all names used are mentioned in declaration statements. INTIALIZATION:

It is useful if a variable can be given an initial value when it is declared. It is bad

practice to rely on the compiler to initialize variables to zero or some other value.
This is not, of course, strictly necessary as a value can always be assigned to a variable. In terms of the security of a language it is important that the compiler checks that a variable is not used before it has had a value assigned to it. The security of languages such as Modula-2 is enhanced by the compiler checking that all variables have been given an initial value. However, a weakness of Modula-2 is that variables cannot be given an initial value when they are declared but have to be initialized explicitly using an assignment statement. CONSTANTS

Some of the entities referenced in a program will have constant values either because they are physical or mathematical entities such as the speed of light or because they are a parameter which is fixed for that particular implementation of the program ,for example the number of control loops being used or the bus address for an input or output device. It is always possible to provide constants by initializing a variable to the appropriate quantity, but this has the disadvantage that it is in secure in that the compiler cannot detect if a further assignment is made which changes the value of the constant. It is also confusing to the reader since there is no indication which entities are constants and which are variables (unless the initial assignment is carefully documented). Pascal provides a mechanism for declaring constants, but since the constant declarations must precede the type declarations, only constants of the predefined types can be declared. This is a severe restriction on the constant mechanism. For example, it is not possible to do the following: TYPE

A Motor State = (OFF, LOW, MEDIUM, HIGH);
CONST
Motor Stop = A Motor State (OFF);

A further restriction in the constant declaration mechanism in Pascal is that the value of the constant must be known at compilation time and expressions are not permitted in constant declarations. The restriction on the use of expressions in constant declarations is removed in Modula-2 (experienced assembler programmers will know the usefulness of being able to use expressions in constant declarations).
For example, in Modula-2 the following are valid constant declarations:
CONST
message = 'a string of characters';

length = 1.6;

breadth = 0.5;

area = length * breadth;

4.3 MODULARITY AND VARIABLES:

Scope and visibility:

The scope of a variable is defined as the region of a program in which the variation is potentially accessible or modifiable. The regions in which it may actually accessed or modified are the regions in which it is said to be visible. Most languages provide mechanisms for controlling scope and visibility. There are two general approaches: languages such as FORTRAN provide a single level locality whereas the block-structured languages such as Modula-2 provide multilevel locality. In the block-structured languages entities which are declared within a block, only be referenced inside that block. Blocks can be nested and the scope extended throughout any nested blocks. This is illustrated in Example which shows scope for a nested PROCEDURE in Modula-2. MODULE ScopeExampLe1;

```
VAR
A, B: INTEGER;
PROCEDURE Level One;
VAR
B, C: INTEGER;
BEGIN
( *
*)
END (* Level one *);
BEGIN
(*
A and B visible here but not Level One and
Level One .C
*)
END ScopeExample1.
```

The *scope* of variables A and B declared in the main module ScopeExample1 extends throughout the program that is they are global variables.

Global and local variables:

Although the compiler can easily handle the reuse of names, it is not as easy for the programmer and the use of deeply nested PRO CEO UR E blocks with the reuse of names can compromise the security of a Pascal or Modula-2 program. As the program shown in Example illustrates the reuse of names can cause confusion as to which entity is being referenced. MODULE ScopeL2;

```
VAR X. Y, Z: INTEGER;
PROCEDURE L 1;
VAR Y: INTEGER;
PROCEDURE L2;
VAR X: INTEGER;
PROCEDURE L3;
VAR Z: INTEGER;
PROCEDURE L4;
BEGIN
Y: = 25; (* L1.Y NOT
LO.Y*) END L4;
BEGIN
(* L1.Y. L2.X, L3.Z visible
*) END L3;
BEGIN
(* L1.Y, L2.X. LO.Z visible
*) END L2;
BEGIN
(* LO.X, L1.Y. LO.Z visible *)
END L 1 ;
BEGIN
(* ••• *)Scope L2.
```

It is very easy to assume in assigning the value 25 to Y in PROCEDURE L4 that the global variable Y is being referenced, when in fact it is the variable Y declared in PROCEDURE L 1 that is being referenced.

4.4 COMPILATION OF MODULAR PROGRAM:

If we have to use a modular approach in designing software how do we compile the modules to obtain executable object code? There are two basic approaches: either combine at the source code level to form a single unit which is then compiled, or compile the individual modules separately and then in some way link the compiled version of each module to form the executable program code. Using the second approach a special piece of software called a linker has to be provided as part of the compilation support to do the linking of the modules. A reason for the popularity and widespread use of FORTRAN for engineering and scientific work is that subroutines can be compiled independently from the main program, and from each other. The ability to carry out compilation independently arises from the single-level scope rules of FORTRAN; the compiler makes the assumption that any entity which is referenced in a subroutine, but not declared within that subroutine, will be declared externally and hence it simply inserts the necessary external linkage to enable the linker to attach the appropriate code. It must be stressed that the compilation is independent that is when a main program is compiled the compiler has no information available which will enable it to check that the reference to the subroutine is correct.

For example, a subroutine may expect three real variables as parameters, but if the user supplies four integer variables in the call statement the error will not be detected by the compiler. Independent compilation of most block-structured languages is even more difficult and prone to errors in that arbitrary restrictions on the use of variables have to be imposed. Many errors can be detected at the linking stage. However, because linking comes later in the implementation process errors discovered at this stage are more costly to correct. It is preferable to design the language and compilation system in such a way as to be able to detect as many errors as possible during compilation instead of when linking. Both Modula-2 and Ada have introduced the idea of separate compilation units. Separate compilation implies that the compiler is provided with some information about the previously or separately compiled units which are to be incorporated into a program. In the case of Modula-2 the source code of the DEFINITION part of a separately compiled module must be made available to the user, and hence the compiler. This enables the compiler to carry out the normal type checking and procedure parameter matching checks. Thus in Modula-2 type mismatches and procedure parameter errors are detectable by the compiler. It also makes available the scope control features of Modula-2. The provision of independent compilation of the type introduced in FORTRAN represented a major advance in supporting software development because it enabled the development of extensive object code libraries. Languages which support separate compilation represent a further advance in that they add greater security and easy error checking to library use.

4.5 DATA TYPES:

As we have seen above, the allocation of types is closely associated with the declaration of entities. The allocation of a type defines the set of values that can be taken by an entity of that type and the set of operations that can be performed on the entity. The richness of types supported by a language and the degree of rigour with which type compatibility is enforced by the language are important influences on the security of programs written in the language. Languages which rigorously enforce type compatibility are said to be *strongly* typed; languages which do not enforce type compatibility are said to be *weakly* typed. FORTRAN and BASIC are weakly typed languages: they enforce some type checking; for example, the statements A $ = 2 5 or A = X$ + Yare not allowed in BASIC, but they allow mixed integer and real arithmetic and provide implicit type changing in arithmetic statements. Both languages support only a limited number of types.

An example of a language which is strongly typed is Modula-2. In addition to enforcing type checking on standard types, Modula-2 also supports enumerated types. The enumerated type allows programmers to define their own types in addition to using the predefined types. Consider a simple motor speed control system which has four settings 0 F F, LOW, ME DIU M, H I GH and which is controlled from a computer system. Using Modula-2 the programmer could make the declarations:

```
TYPE
AMotorState = (OFF, LOW, MEDIUM, HIGH);
VAR
motor Speed: AMotorState;
```

The variable motor Speed can be assigned only one of the values enumerated in the T YP E definition statement. An attempt to assign any other value will be trapped by the compiler, for example the statement will be flagged as an error.

If we contrast this with the way in which the system could be programmed
using FORTRAN we can see some of the protection which strong typing
provides. In ANSI FORTRAN integers must be used to represent the four states
of the motor Control:

INTEGER OFF, LOW, MEDIUM, HIGH

DATA OFF/0/, LOW/1/, MEDIUM/2/, HIGH/3/

If the programmer is disciplined and only uses the defined integers to set MSPEED then the program is clear and readable, but there is no mechanism to prevent direct assignment of any value to MS PEE D.

Hence the statements

MSPEED = 24

MSPEED = 1 SO

would be considered as valid and would not be flagged as errors either by the compiler or by the run-time system. The only way in which they could be detected is if the programmer inserted some code to check the range of values before sending them to the controller. In FORTRAN a programmer-inserted check would be necessary since the output of a value outside the range 0 to 3 may have an unpredictable effect on the motor speed.

4.6 EXCEPTION HANDLING:

One of the most difficult areas of program design and implementation is the handling of errors, unexpected events (in the sense of not being anticipated and hence catered for at the design stage) and exceptions which make the processing of data by the subsequent segments superfluous, or possibly dangerous. The designer has to make decisions on such questions as what errors are to be detected. What sort of mechanism is to be used to do the detection? And what should be done when an error is detected? Most languages provide some sort of automatic error detection mechanisms as part of their run-time support system. Typically they trap errors such as an attempt to divide by zero, arithmetic overflow, array bound violations, and sub-range violations; they may also include traps for input/output errors. For many of the checks the compiler has to add code to the program; hence the checks increase the size of the code and' reduce the speed at which it executes. In most languages the normal response when an error is detected is to halt the program and display an error message on the user's terminal. In a development environment it may be acceptable for a program to halt following an error; in a real-time system halting the program is not acceptable as it may compromise the safety of the system. Every attempt must be made to keep the system running.

4.7 LOW LEVEL FACILITIES:

In programming real-time systems we frequently need to manipulate directly data in specific registers in the computer system, for example in memory registers, CPU registers and registers in an input! output device. In the older, high-level languages, assembly-coded routines are used to do this. Some languages provide extensions to avoid the use of assembly routines and these typically are of the type found in many versions of BASIC. These take the following form: PEEK (address) - returns as INTEGER variable contents of the location address.

POKE (address, value) - puts the INTEGER value in the location address.

It should be noted that on eight-bit computers the integer values must be in the range o to 255 and on 16 bit machines they can be in the range 0 to 65 535. For computer systems in which the input/output devices are not memory mapped, for example Z80 systems, additional functions are usually provided such as INP (address) and OUT (address, value). A slightly different approach has been adopted in BBC BASIC which uses an 'indirection' operator. The indirection operator indicates that the variable which follows it is to be treated as a pointer which contains the address of the operand rather than the operand itself (the term indirection is derived from the indirect addressing mode in assembly languages). Thus in BBC BASIC the following code

```
100 DACAddress=&FE60
120? DACAddress=&34
```

results in the hexadecimal number 34 being loaded into location FE 60 H; the indirection operator is '?'. In some of the so-called Process FORTRAN languages and in CORAL and

RTL/2 additional features which allow manipulation of the bits in an integer variable are provided, for example

SETBITJ (I),

IF BIT J(I) n1 ,n2 (where I refers to the bit In

variable.

Also available are operations such as AND, 0 R, S LA, S RA, etc., which mimic the operations available at assembly level. The weakness of implementing low-level facilities in this way is that all type checking is lost and it is very easy to make mistakes. A much more secure method is to allow the programmer to declare the address of the register or memory location and to be able to associate a type with the declaration, for example

which declares a variable of type CHAR located at memory location 0 FE60 H.

Characters can then be written to this location by simple assignment

Modula-2 provides a low-level support mechanism through a simple set of primitives which have to be encapsulated in a small nucleus coded in the assembly language of the computer on which the system is to run. Access to the primitives is through a module SYS TEM which is known to the compiler. SYST EM can be thought of as the software bus linking the nucleus to the rest of the software modules. SYSTEM makes available three data types, WORD, ADDRESS, PROCESS, and six procedures, ADR, SIZE, TSIZE, NEWPROCESS, TRANSFER, I 0 TRANS FE R. W0 RD is the data type which specifies a variable which maps onto one unit of the specific computer storage. As such the number of bits in a WORD will vary from implementation to implementation; for example, on a PDP·II implementation a WORD is 16 bits, but on a 68000 it would be 32 bits. ADDRESS corresponds to the definition TYPEA DDRES S = POI NTER TOW0 RD, that is objects of type ADDRES S are pointers to memory units and can be used to compute the addresses of memory words. Objects of type PROC ESS have associated with them storage for the volatile environment of the particular computer on which Modula-2 is implemented; they make it possible to create easily process (task) descriptors. Three of the procedures provided by SYSTEM are for address manipulation:

FROM S

AD

EXPOR

ADR (v) returns the ADDRESS of variable v SIZE

(v) returns the SIZE of variable v in WORDs TSIZE

(t) returns the SIZE of any variable of type t

inWORDs.

In addition variables can be mapped onto specific memory locations. This facility can be used for writing device driver modules in Modula-2. A combination of the low-level access facilities and the module concept allows details of the hardware device to be hidden within a module with only the procedures for accessing the module being made available to the end user.

4.8 CO ROUTINES:

In Modula-2 the basic form of concurrency is provided by co routines. The two procedures NEW PRO C E S sand T RAN S FE R exported by S Y S T EM are defined as follows: PROCEDURE NEWPROCESS (ParameterLessProcedure: PROC);
workspace Address: ADDRESS;
workspace Size: CARDINAL;
VAR co routine: ADDRESS (* PROCESS *));
PROCEDURE TRANSFER (VAR source, destination:
ADDRESS (*PROCESS*));
Any parameter less procedure can be declared as a PROCESS. The procedure NEW PRO C E S S S associates with the procedure storage for the process parameters The amount to be allocated depends on the number and size of the variables local to the procedure forming the coroutine, and to the procedures which it calls. Failure to allocate sufficient space will usually result in a stack overflow error at run-time. The variable co routine is initialized to the address which identifies the newly created co routine and is used as a parameter in calls to T RAN S FER. The transfer of control between co routines is made using a standard procedure T RAN SF ER which has two arguments of type ADD RES S (PROCESS) . The first is the calling co routine and the second is the co routine to which control is to be transferred. The mechanism is illustrated in Example 5.13. In this example the two parameter less procedures form the two co routines which pass control to each other so that the message

Co routine one and Co routine two

is printed out 25 times. At the end of the loop, Co routine 2 passes control
back to Main Program.

CONCURRENCY:

Wirth (1982) defined a standard module Processes s which provides a higher-level mechanism than co routines for concurrent programming. The module makes no assumption as to how the processes (tasks) will be implemented; in particular it does not assume that the processes will be implemented on a single processor.

4.9 OVERVIEW OF REAL-TIME:

The best way to start an argument among a group of computer scientists, software engineers or systems engineers is to ask them which is the best language to use for writing software. Rational arguments about the merits and demerits of any particular language are likely to be submerged and lost in a sea of prejudice. Since 1970 high-level languages for the programming and construction of real time systems have become widely available. Early languages include: CORAL (Woodward et a/., 1970) and RTL/2 (Barnes, 1976) as well as modifications to FORTRAN and BASIC. More recently the interest in concurrency and multiprocessing has resulted in many languages with the potential for use with real-time systems. These include Ada (see Young, 1982; Burns and Wellings, 1990), ARGUS (Liskovand Scheifler, 1983), CONIC (Kramer et a/., 1983), CSP (Hoare, 1978), CUTLASS (CEGB, see Bennett and Linkens, 1984), FORTH (Brodie, 1986),

A language suitable for programming real-time and distributed systems must have all the characteristics of a good, modern, non-real-time language; that is it should have a clean syntax, a rational procedure for declarations, initialisation and typing of variables, simple and consistent control structures, clear scope and visibility rules, and should provide support for modular construction. The addition required for real-time use includes support for concurrency or multi-tasking and mechanisms to permit access to the basic computer functions (usually referred low-level constructs).

Recommended question:

1. In the computer science literature you will find lots of arguments about 'global' and 'Local' variables. What guidance would you give to somebody who asked for advice on how to decide on the use of global or local variables?
2. Why is it useful to have available a predefined data type BITSET in Modula-2? Give an example to illustrate how, and under what circumstances, BITSET would be used.
3. How does strong data typing contribute to the security of a programming language?
4. Explain simple table-driven approach used for application oriented software.
5. **W**hat are .the major requirements for CUTLASS? Explain in detail, with host-target

Configuration.

6. How do strong data typing contribute to the security of programming language?
7. What are the requirements, which CUTLASS has to meet? With a neat diagram, show the CUTLASS host-target configuration.

Module-4

Operating Systems

Introduction, Real –Time Multi –Tasking OS, Scheduling Strategies, Priority Structures, Task Management, Scheduler and Real –Time Clock Interrupt Handles, Memory Management ,Code Sharing, Resource control, Task Co-operation and Communication, Mutual Exclusion, Data Transfer, Liveness, Minimum OS Kernel, Examples.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.
3. **Real time Systems Development**, Rob Williams, Elsevier, 2006.

5.1 OPERATING SYSTEMS

INTRODUCTION

Specific computer using a particular language can be hidden from the designer. An operating system for a given computer converts the hardware of the system into a virtual machine with characteristics defined by the operating system. Operating systems were developed, as their name implies, to assist the operator in running a batch processing computer; they then developed to support both real-time systems and multi-access on-line systems. The traditional approach is to incorporate all the requirements inside a general purpose operating system as illustrated in Figure 6.1. Access to the hardware of the system and to the I/O devices is through the operating system. In many real-time and multi-programming systems restriction of access is enforced by hardware and software traps.

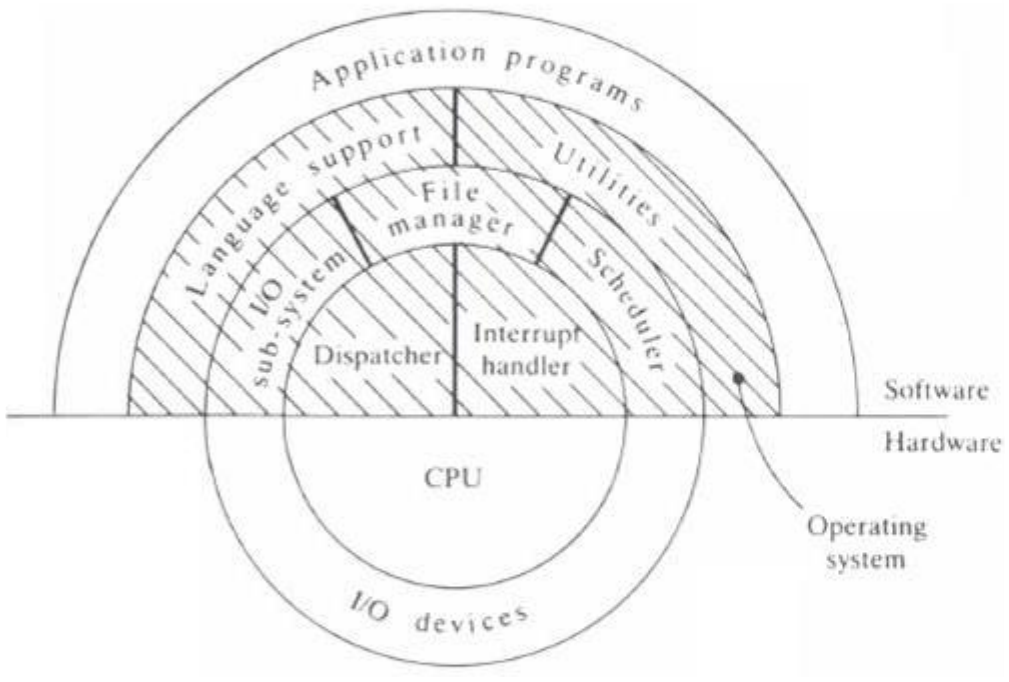


Figure 6.1: General purpose operating system.

The operating system is constructed, in these cases, as a monolithic monitor. In single-job operating systems access through the operating system is not usually enforced; however, it is good programming practice and it facilitates portability since the operating system entry points remain constant across different implementations. In addition to supporting and controlling the basic activities, operating systems provide various utility programs, for example loaders, linkers, assemblers and debuggers, as well as run-time support for high-level languages.

A general purpose operating system will provide some facilities that are not required in a particular application, and to be forced to include them adds unnecessarily to the system overheads. Usually during the installation of an operating system certain features can be selected or omitted. A general purpose operating system can thus be 'tailored' to meet a specific application requirement. Recently operating systems which provide only a minimum kernel or nucleus have become popular; additional features can be added by the applications programmer writing in a high-level language. This structure is shown in Figure 6.2. In this type of operating system the distinction between the operating system and the application software becomes blurred. The approach has many advantages for applications that involve small, embedded systems.

5.2 REAL-TIME MULTI-TASKING OS:

There are many different types of operating systems and until the early 1980s there was a clear distinction between operating systems designed for use in real-time applications and other types of operating system. In recent years the dividing line has become blurred. For example, languages such as Modula-2 enable us to construct multi-tasking real-time applications that run on top of single-user, single· task operating systems. And operating systems such as UNIX and OS/2 support multi-user, multi-tasking applications. Confusion can arise between multi-user or multi-programming operating systems and multi-tasking operating systems. The function of a multi-user operating system is illustrated in Figure 6.4: the operating system ensures that each user can run a single program as if they had the whole of the computer system for their program.

Although at any given instance it is not possible to predict which user will have the use of the CPU, or even if the user's code is in the memory, the operating system ensures that one user program cannot interfere with the operation of another user program. Each user program runs in its own protected environment. A primary concern of the operating system is to prevent one program, either deliberately or through error, corrupting another. In a multi-tasking operating system it is assumed that there is a single user and that the various tasks co-operate to serve the requirements of the user. Co-operation requires that the tasks communicate with each other and share common data. This is illustrated in Figure 6.5. In a good multitasking operating system task communication and data sharing will be regulated so that the operating system is able to prevent inadvertent communication or data access (that is, arising through an error in the coding of one task) and hence protect data which is private to a task (note that deliberate interference cannot be prevented the tasks are assumed to be co-operating).

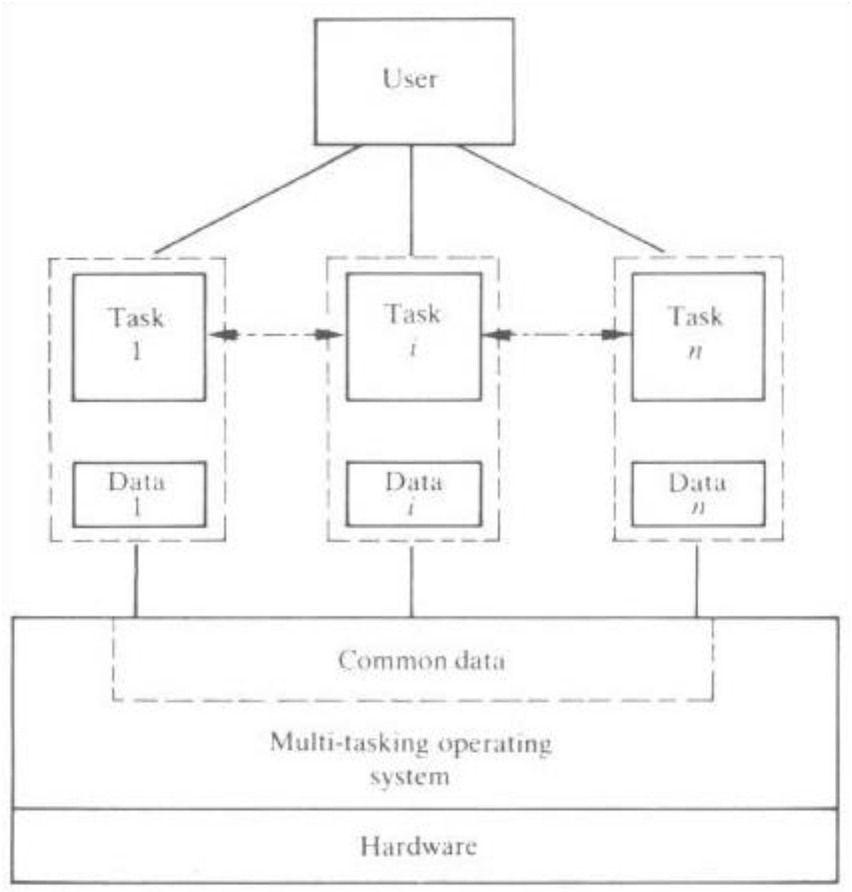


Figure: Multitasking operating system.

A real-time multi-tasking operating system has to support the resource sharing and the timing requirements of the tasks and the functions can be divided as follows:

Task management: the allocation of memory and processor time (scheduling) to tasks.

Memory management: control of memory allocation.

Resource control: control of all shared resources other than memory and CPU time.

Intertask communication and synchronization: provision of support mechanisms to provide safe communication between tasks and to enable tasks to synchronize their activities.

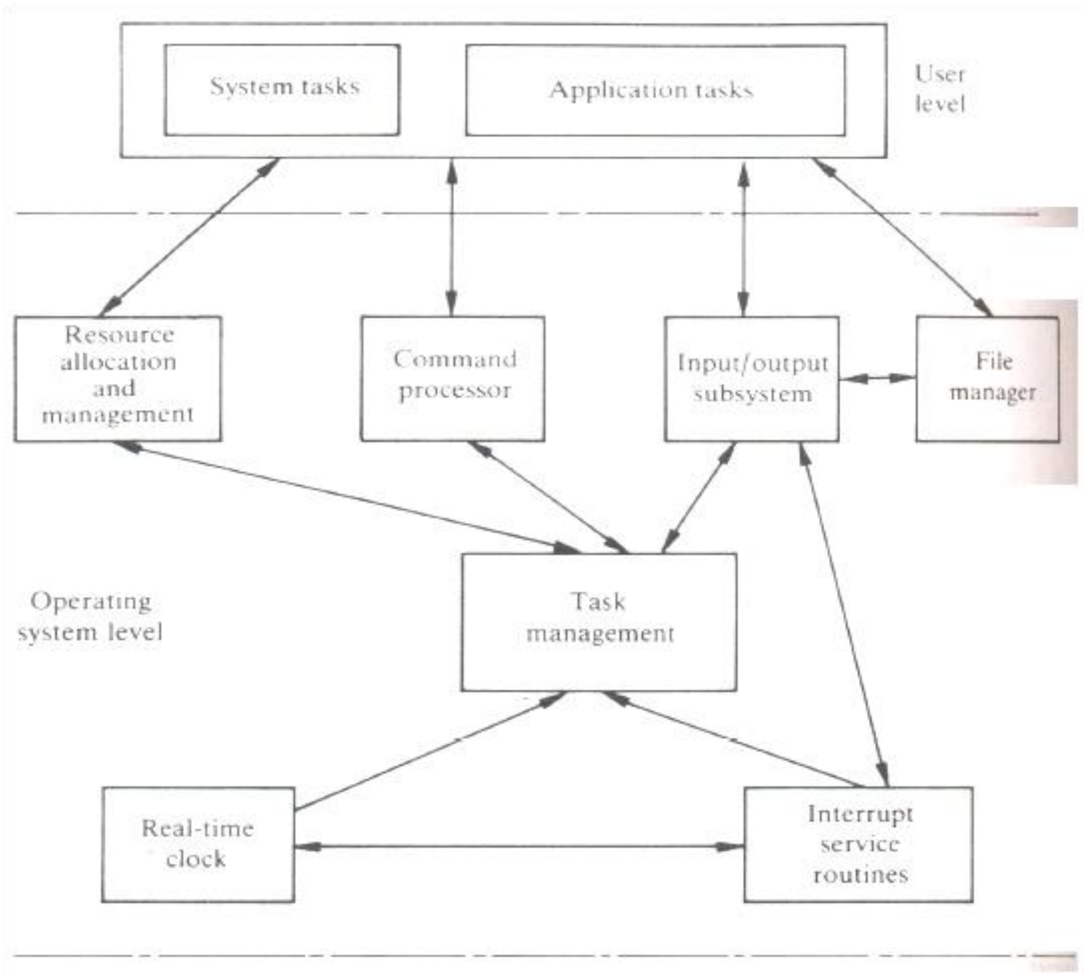


Figure: Typical structure of real-time operating system.

5.3 SCHEDULING STRATEGIES:

If we consider the scheduling of time allocation on a single CPU there are two basic strategies:

1. Cyclic.
2. Pre-emptive.

1. Cyclic

The first of these, cyclic, allocates the CPU to a task in turn. The task uses the CPU for as long as it wishes. When it no longer requires it the scheduler allocates it to the next task in the list. This is a very simple strategy which is highly efficient in that it minimizes the time lost in switching between tasks. It is an effective strategy for small embedded 'systems for which the execution times for each task run are carefully calculated (often by counting the number of machine instruction cycles for. the task) and for which the software is carefully divided into appropriate task segments. In general this approach is too restrictive since it requires that the task units have similar execution times. It is also difficult to deal with random events using this method.

2. Pre-emptive.

There are many pre-emptive strategies. All involve the possibility that a task will be interrupted - hence the term pre-emptive - before it has completed a particular invocation. A consequence of this is that the executive has to make provision to save the volatile environment for each task, since at some later time it will be allocated CPU time and will want to continue from the exact point at which it was interrupted. This process is called context switching and a mechanism for supporting it is described below. The simplest form of pre-emptive scheduling is to use a time slicing approach (sometimes called a round-robin method). Using this strategy each task is allocated a fixed amount of CPU time - a specified number of ticks of the clock – and at the end of this time it is stopped and the next task in the list is run. Thus each task in turn is allocated an equal share of the CPU time. If a task completes before the end of its time slice the next task in the list is run immediately.

The majority of existing RTOSs use a priority scheduling mechanism. Tasks are allocated a priority level and at the end of a predetermined time slice the task with the highest priority of those ready to run is chosen and is given control of the CPU. Note that this may mean that the task which is currently running continues to run. Task priorities may be fixed - a static priority system - or may be changed during system execution - a dynamic priority system. Dynamic priority schemes can increase the flexibility of the system, for example they can be used to increase the priority of particular tasks under alarm conditions. Changing priorities is, however, risky as it makes it much harder to predict the behavior of the system and to test it. There is the risk of locking out certain tasks for long periods of time. If the software is well designed and there is adequate computing power there should be no need to change priorities - all the necessary constraints will be met. If it is badly designed and/or there are inadequate computing resources then dynamic allocation of priorities will not produce a viable, reliable system.

5.4 PRIORITY STRUCTURES:

In a real-time system the designer has to assign priorities to the tasks in the system. The priority will depend on how quickly a task will have to respond to a particular event. An event may be some activity of the process or may be the elapsing of a specified amount of time.

1. Interrupt level: at this level are the service routines for the tasks and devices which require very fast response - measured in milliseconds. One of these tasks will be the real-time clock task and clock level dispatcher.
2. Clock level: at this level are the tasks which require repetitive processing, such as the sampling and control tasks, and tasks which require accurate timing. The lowest-priority task at this level is the base level scheduler.
3. Base level: tasks at this level are of low priority and either have no deadlines to meet or are allowed a wide margin of error in their timing. Tasks at this level may be allocated priorities or may all run at a single priority level - that of the base level scheduler.

Interrupt level:

As we have already seen an interrupt forces a rescheduling of the work of the CPU and the system has no control over the timing of the rescheduling. Because an interrupt-generated rescheduling is outside the control of the system it is necessary to keep the amount of processing to be done by the interrupt handling routine to a minimum. Usually the interrupt handling routine does sufficient processing to preserve the necessary information and to pass this information to a further handling routine which operates at a lower-priority level, either clock level or base level. Interrupt handling routines have to provide a mechanism for task swapping, that is they have to save the volatile environment.

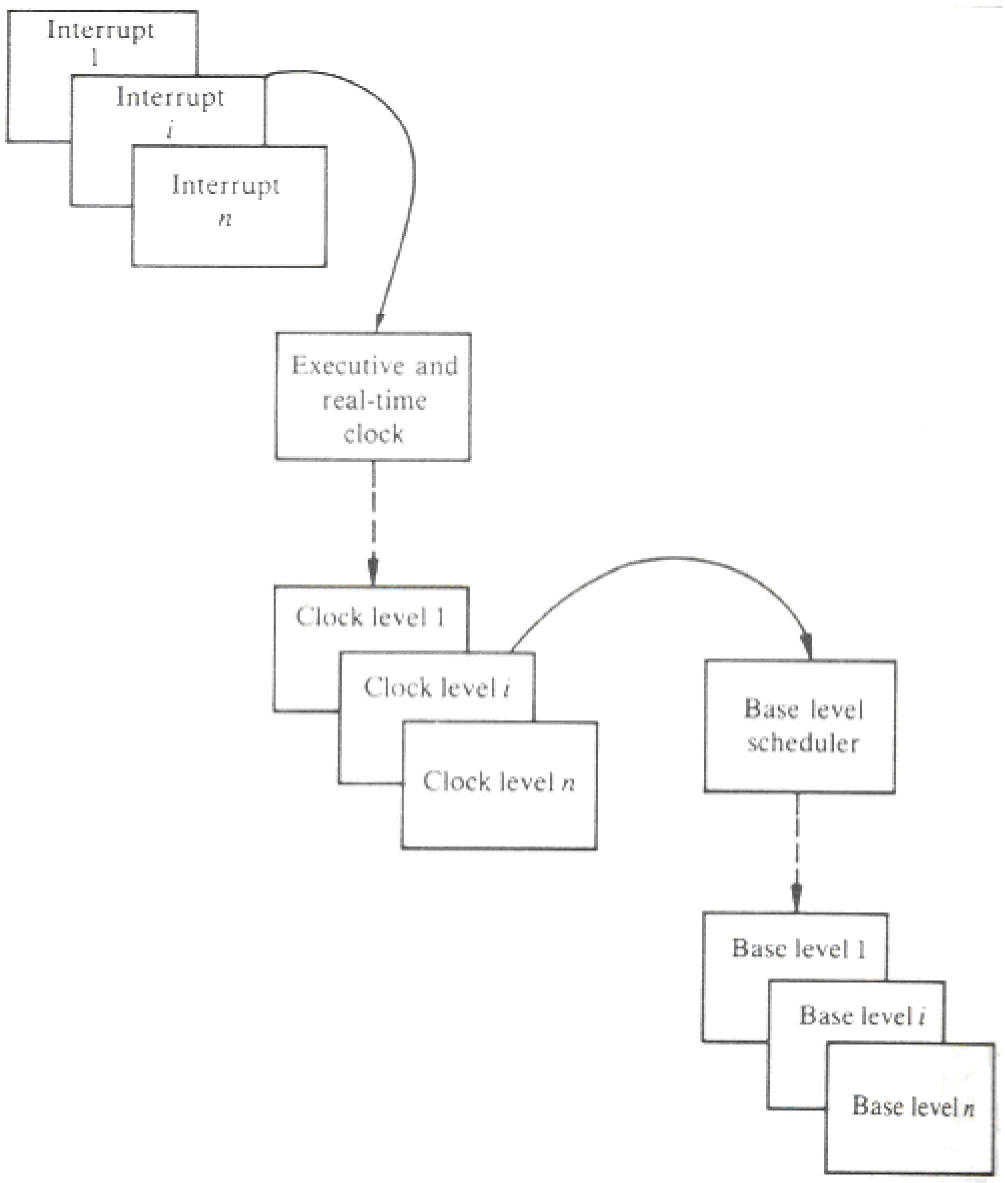


Figure: Priority levels in an RTOS.

Clock level:

One interrupt level task will be the real-time clock handling routine which will be entered at some interval, usually determined by the required activation rate for the most frequently required task. Typical values are I to 200 ms. Each clock interrupt is known as a *tick* and represents the smallest time interval known to the system. The function of the clock interrupt handling routine is to update the time-of-day clock in the system and to transfer control to the dispatcher. The scheduler selects which task is to run at a particular clock tick. Clock level tasks divide into two categories:

1. CYCLIC: these are tasks which require accurate synchronization with the outside world.

2. DELA Y: these tasks simply wish to have a fixed delay between successive repetitions or to delay their activities for a given period of time.

Cyclic tasks:

The *cyclic* tasks are ordered in a priority which reflects the accuracy of timing required for the task, those which require high accuracy being given the highest priority. Tasks of lower priority within the clock level will have some jitter since they will have to await completion of the higher - level tasks.

Delay tasks:

The tasks which wish to delay their activities for a fixed period of time, either to allow some external event to complete (for example, a relay may take 20 ms to close) or because they only need to run at certain intervals (for example, to update the operator display), usually run at the base level. When a task requests a delay its status is changed from runnable to suspended and remains suspended until the delay period has elapsed.

One method of implementing the delay function is to use a queue of task descriptors, say identified by the name DELAYED. This queue is an ordered list of task descriptors, the task at the front of the queue being that whose next running time is nearest to the current time.

Base level:

The tasks at the base level are initiated on demand rather than at some predetermined time interval. The demand may be user input from a terminal, some process event or some particular requirement of the data being processed. The way in which the tasks at the base level are scheduled can vary; one simple way is to use time slicing on a round-robin basis. In this method each task in the runnable queue is selected in turn and allowed to run until either it suspends or the base level scheduler is again entered. For real-time work in which there is usually some element of priority this is not a particularly satisfactory solution. It would not be sensible to hold up a task, which had been delayed waiting for a relay to close but was now ready to run, in order to let the logging task run.

Most real-time systems use a priority strategy even for the base level tasks. This may be either a fixed level of priority or a variable level. The difficulty with a fixed level of priority is in determining the correct priorities for satisfactory operation; the ability to change priorities dynamically allows the system to adapt to particular circumstances. Dynamic allocation of priorities can be carried out using a high-level scheduler or can be done on an *ad hoc* basis from within specific tasks. The high level scheduler is an operating system task which is able to examine the use of the system resources; it may for example check how long tasks have been waiting and increase the priority of the tasks which have been waiting a long time. The difficulty with the high-level scheduler is that the algorithms used can become complicated and hence the overhead in running can become significant.

5.5 TASK MANAGEMENT:

The basic functions of the task management module or executive are:

1. To keep a record of the state of each task;
2. To schedule an allocation of CPU time to each task; and
3. To perform the context switch, that is to save the status of the task that is currently using the CPU and restore the status of the task that is being allocated CPU time.

In most real-time operating systems the executive dealing with the task management functions is split into two parts: a scheduler which determines which task is to run next and which keeps a record of the state of the tasks, and a dispatcher which performs the context switch. Task states:

With one processor only one task can be running at any given time and hence the other tasks must be in some other state. The number of other states, the names given to the states, and the transition paths between the different states vary from operating system to operating system. A typical state diagram is given in Figure6.1 and the various states are as follows (names in parentheses are commonly are. alternatives):

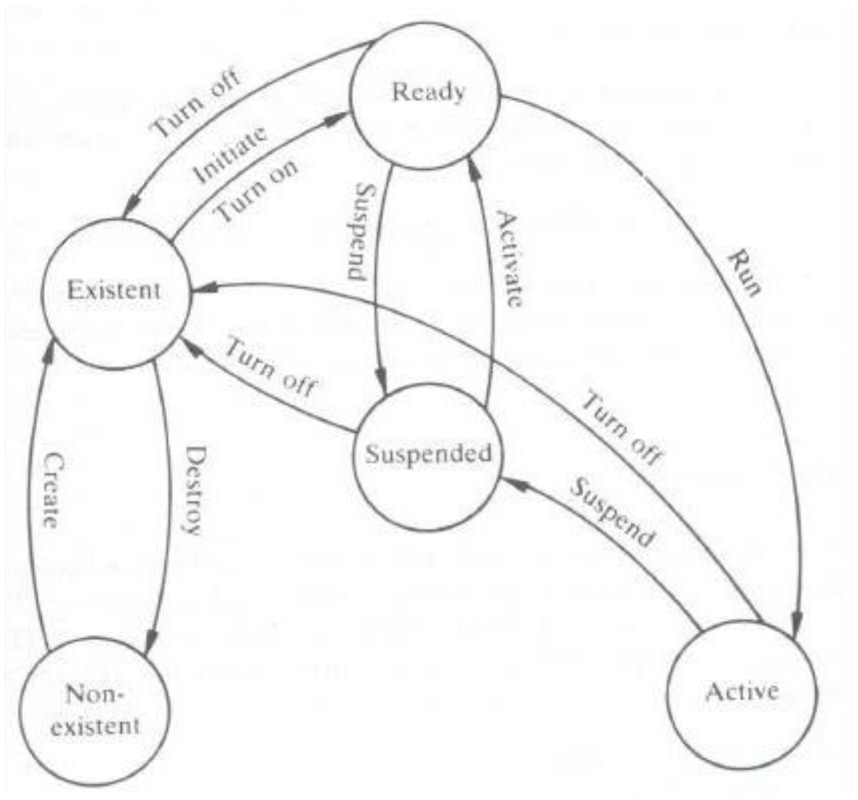


Figure: Example of a typical task state diagram.

• Active (running): this is the task which has control of the CPU. It will normally be the task with the highest priority of the tasks which are ready to run.

• Ready (runnable, on): there may be several tasks in this state. The attribute of the task and the resources required to run the task must be available for the task to be placed in the Ready state.

• Suspended (waiting, locked out, delayed): the execution of tasks placed this state has been suspended because the task requires some resource which is not available or because the task is waiting for some signal from the plant for example input from the analog-to-digital converter, or because the task is waiting for the elapse of time.

• Existent (dormant, off): the operating system is aware of the existence of this task, but the task has not been allocated a priority and has not been made runnable.

• Non-existent (terminated): the operating system has not as yet been made aware of the existence of this task, although it may be resident in the. memory of the computer.

Task descriptor:

Information about the status of each task is held in a block of memory by the RTOS. This block is referred to by various names· task descriptor (TD), process descriptor (PD), task control block (TCB) or task data block (TDB). The information held in the TD will vary from system to system, but will typically consist of the following:

- Task identification (10);
- Task priority (P);
- Current state of task;
- Area to store volatile environment (or a pointer to an area for storing the volatile environment); and
- Pointer to next task in a list.

5.6 SCHEDULER AND REAL-TIME CLOCK INTERRUPT HANDLES:

The real-time clock handler and the scheduler for the clock level tasks must be carefully designed as they run at frequent intervals. Particular attention has to be paid to the method of selecting the tasks to be run at each clock interval. If ached of all tasks were to be carried out then the overheads involved could become significant.

System commands which change task status.

The range of system commands affecting task status varies with the operating system. Typical states and commands are shown in Figure 6.12 and fuller details of the commands are given in Table. Note that this system distinguishes between tasks which are suspended awaiting the passage of time - these tasks are marked as delayed - and those tasks which are waiting for an event or a system resource these are marked as locked out. The system does not explicitly support base level tasks; however, the lowest four priority levels of the clock level tasks can be used to create a base level system A so - called free time executive (FTX) is provided which if used runs at priority level n - 3 where n is the lowest-priority task number. The FTX is used to run tasks at priority levels n - 2, n - I and n; it also provides support for the chaining of tasks. The dispatcher is unaware of the fact that tasks at these three priority levels are being changed; it simply treats whichever tasks are in the lowest three priority levels as low-priority tasks. Tasks run under the FTX do not have access to the system commands (except OFFCO1 that is turn task off).

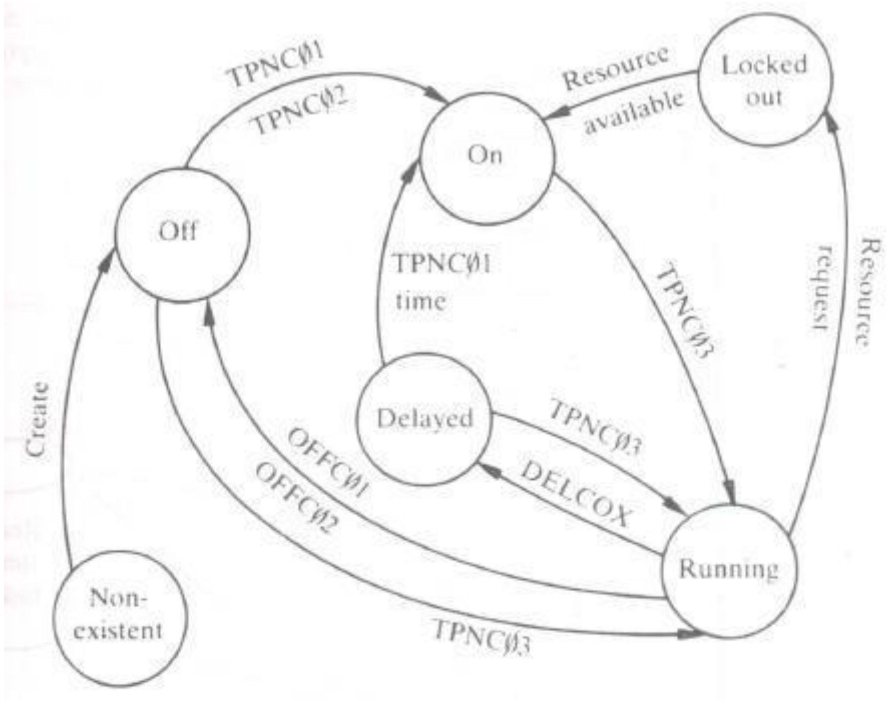


Figure: RTOS state diagram.

Dispatcher- search for work:

The dispatcher/scheduler has two entry conditions:

1. The real-time clock interrupt and any interrupt which signals the completion of an input/output request;
2. A task suspension due to a task delaying, completing or requesting an input/output transfer.

In response to the first condition the scheduler searches for work starting with the highest-priority task and checking each task in priority order (see Figure 6.14). Thus if tasks with a high repetition rate are given a high priority they will be treated as if they were clock level tasks, that is they will be run first during each system clock period. In response to the second condition a search for work is started at the task with the next lowest priority to the task which has just been running. There cannot be another higher-priority task ready to run since a higher-priority task becoming ready always pre-empts a lower-priority-running task. The system commands for task management are issued as calls from the assembly level language and the parameters are passed either in the CPU registers or as a control word immediately following the call statement.

5.7 MEMORY MANAGEMENT:

Since the majority of control application software is static - the software is not dynamically created or eliminated at run-time - the problem of memory management is simpler than for multi-programming, on-line systems. Indeed with the cost of computer hardware, both processors and memory, reducing many control applications use programs which are permanently resident in fast access memory. With permanently resident software the memory can be divided as shown in Figure. The user space is treated as one unit and the software is linked and loaded as a single program into the user area. The information about the various tasks is conveyed to the operating system by means of a create task statement. Such a statement may be of the form the exact form of the statement will depend on the interface between the high-level language and the operating system. An alternative arrangement is shown in Figure. The available memory is divided into predetermined segments and the tasks are loaded individually into the various segments. The load operation would normally be carried out using to command processor. With this type of system the entries in the TD (or the operation system tables) have to be made from the console using a memory examine as change facility.

Divided (partitioned) memory was widely used in many early real-time operating systems and it was frequently extended to allow several tasks to share on:

partition; the tasks were kept on the backing store and loaded into the appropriate partition when required. There was of course a need to keep any tasks in which timing was crucial (hard time constraint tasks) in fast access memory permanent other tasks could be swapped between fast memory and backing store. The difficulty with this method is, of course, in choosing the best mix of partition sizes. The partition size and boundaries have to be determined at system generation.

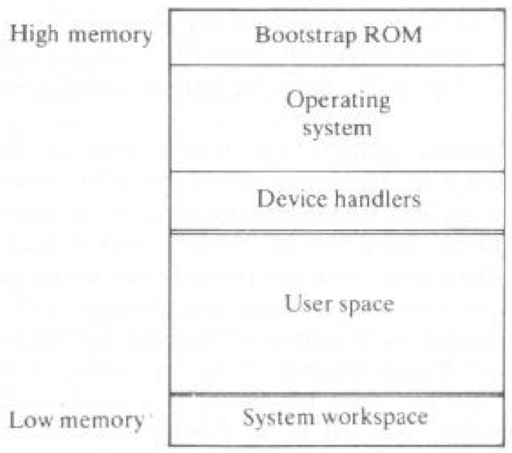


Figure: Non-partitioned memory.

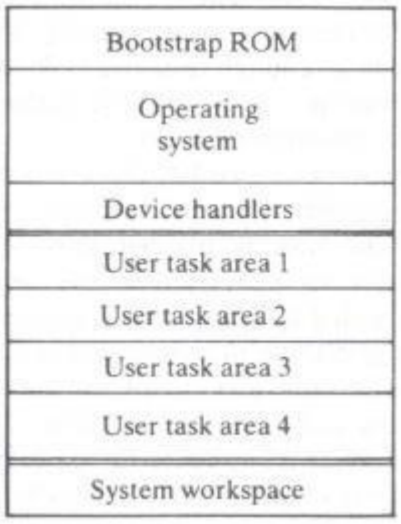


Figure: Partitioned memory.

5.8 CODE SHARING:

In many applications the same actions have to be carried out in several different tasks. In a conventional program the actions would be coded as a subroutine and one copy of the subroutine would be included in the program. In a multi-tasking system each task must have its own copy of the subroutine or some mechanism must be provided to prevent one task interfering with the use of the code by another task. The problems which can arise are illustrated in Figure 6.20. Two tasks share the subroutine S. If task A is using the subroutine but before it finishes some even occurs which causes a rescheduling of the tasks and task B runs and uses the subroutine, then when a return is made to task

A, although it will begin to use subroutine S again at the correct place, the values of locally held data will have been changed and will reflect the information processed within the subroutine by task B. Two methods can be used to overcome this problem:

- serially reusable code; and
- re-entrant code.

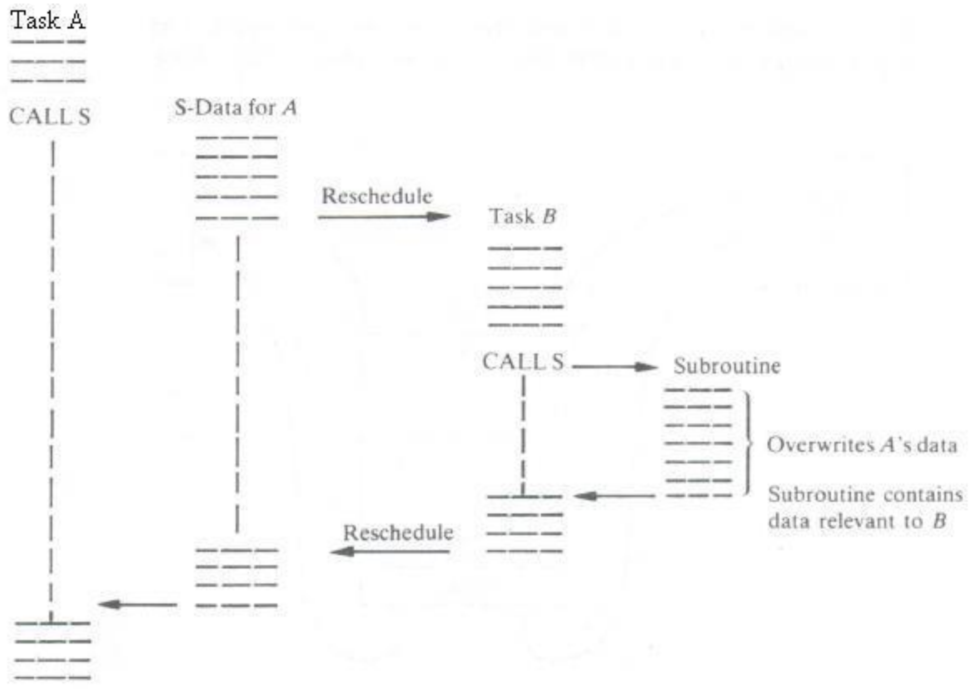


Figure: Sharing a subroutine in multi-tasking system.

Serially reusable code:

As shown in Figure, some form of lock mechanism is placed at the beginning of the routine such that if any task is already using the routine the calling task will not be allowed entry until the task which is using the routine unlocks it. The use of a lock mechanism to protect a subroutine is an example of the need for mechanisms to support mutual exclusion when constructing an operating system.

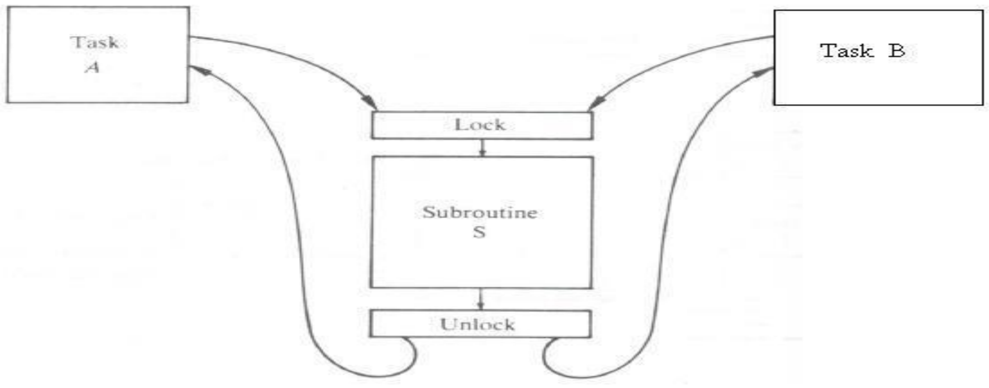


Figure: Serially reusable code.

Re-entrant code:

If the subroutine-can be coded such that it does not hold within it any data that is it is purely code - any intermediate results are stored in the calling task or in a stack associated with the task - then the subroutine is said to be re-entrant. Figure shows an arrangement which can be used: the task descriptor for each task contains a pointer to a data area - usually a stack area - which is used for the storage of all information relevant to that task when using the subroutine. Swapping between tasks while they are using the subroutine will not now cause any problems since the contents of the stack pointer will be saved with the volatile environment of the task and will be restored when the task resumes.

All accesses to data by the subroutine will be through the stack and hence it will automatically manipulate the correct data. Re-entrant routines can be shared between several tasks since they contain no data relevant to a particular task and hence can be stopped and restarted at a different point in the routine without any loss of information. The data held in the working registers of the CPU is stored in the relevant task descriptor when task swapping takes place. Device drivers in conventional operating systems are frequently implemented using re-entrant code. The PID control1er code segment uses the information in the LOOP descriptor and the T ASK to calculate the control value and to send it to the control1er. The actual task is made up of the LOOP descriptor, the TASK segment and the PID control code segment. The addition of another loop to the system requires the provision of new loop descriptors; the actual PID control code remains unchanged.

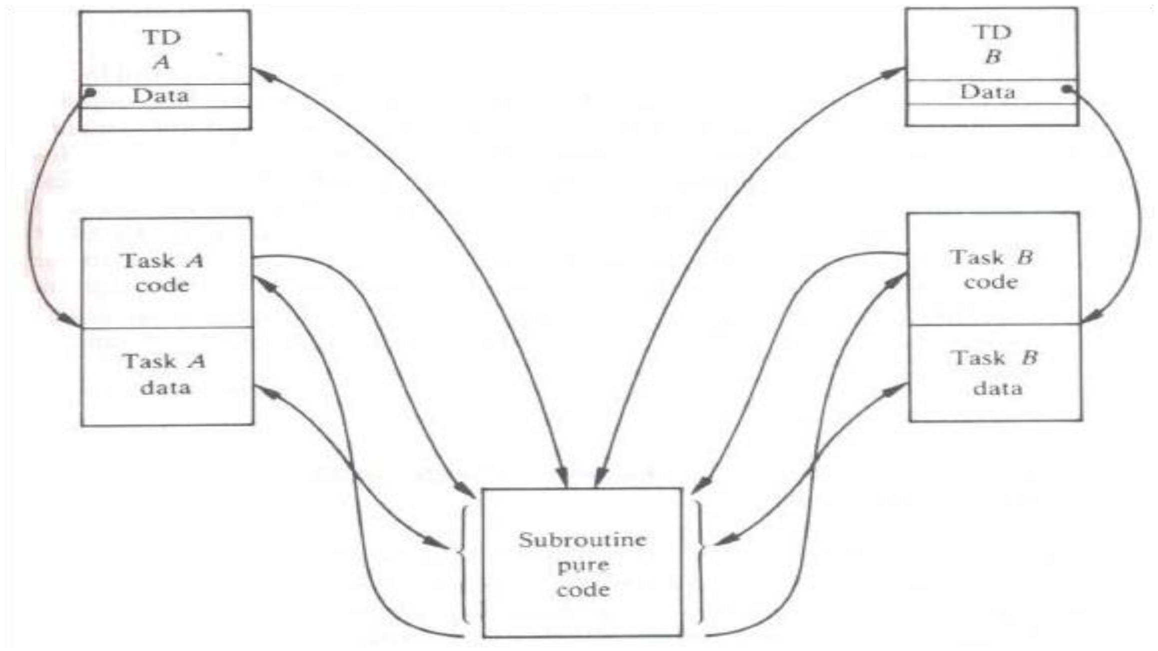


Figure: Use of re-entrant code for sharing.

5.9 RESOURCE CONTROL: AN EXAMPLE OF AN INPUT/OUTPUT SUBSYSTEM (lOSS)

One of the most difficult areas of programming is the transfer of information to and from external devices. The availability of a wel1-designed and implemented input/output subsystem (lOSS) in an operating system is essential for efficient programming. The lOSS handles all the details of the devices. In a multi-tasking system the lOSS should also deal with all the problems of several tasks attempting to access the same device. A typical lOSS will be divided into two levels as shown in Figure. The I/O manager accepts the system calls from the user tasks and transfers the information contained in the calls to the device control block (DCB) for the particular device.

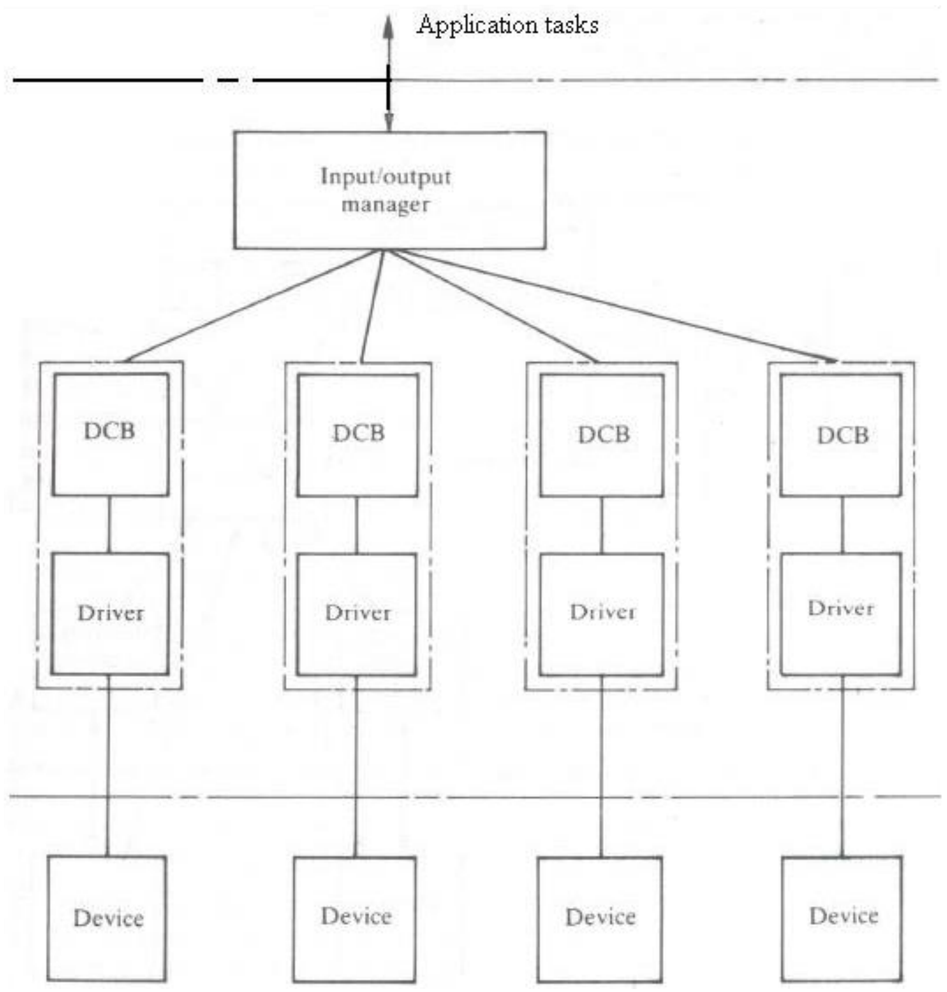


Figure: General structure of IOSS.

5.10 TASK CO-OPERATION AND COMMUNICATION:

In real-time systems tasks are designed to fulfil a common purpose and hence they need to communicate with each other. However, they may also be in competition for the resources of the computer system and this competition must be regulated. Some of the problems which arise have already been met in considering the input/output subsystem and they involve:

- Mutual exclusion;
- Synchronization; and
- Data transfer.

Mutual exclusion:

A multi-tasking, operating system allows the sharing of resources between several concurrently active tasks. This does not imply that the resources can be used simultaneously. The use of some resources is restricted to only one task at a time. For others, for example a re-entrant code module, several tasks can be using them at the same time. The restriction to one task at a time has to be made for resource such as input and output devices; otherwise there is a danger that input intended for one task could get corrupted by input for another task. Similarly problems can arise if two tasks share a data area and both tasks can write to the data area.

Data transfer:

RTOSs typically support two mechanisms for the transfer or sharing of data between tasks: these are the pool and the channel.

Pool is used to hold data common to several tasks, for example tables of values or parameters which tasks periodically consult or update. The write operation on a pool is destructive and the read operation is non-destructive.

Channel supports communication between producers and consumers of data. It can contain one or more items of information. Writing to a channel adds an item without changing items already in it. The read operation is destructive in that it removes an item from the channel. A channel can become empty and also, because in practice its capacity is finite, it can become full.

It is normal to create a large number of pools so as to limit the use of global common data areas. To avoid the problem of two or more tasks accessing a pool simultaneously mutual exclusion on pools is required. The most reliable form of mutual exclusion for a pool is to embed the pool inside a monitor. Given that the read operation does not change the data in a pool there is no need to restrict read access to a pool to one task at a time. Channels provide a direct communication link between tasks, normally on a one-to-one basis. The communication is like a pipe down which successive collections of items of data - messages - can pass. Normally they are implemented so that they can contain several messages and so they act as a buffer between the tasks. One task is seen as the *producer* of information and the other as the *consumer*. Because of the buffer function of the channel the producer and consumer tasks can run asynchronously. There are two basic implementation mechanisms for a channel:

- Queue (linked list); and
- Circular buffer.

The advantage of the queue is that the number of successive messages held in the channel is not fixed. The length of the queue can grow, the only limit being the amount of available memory. The disadvantage of the queue is that as the length of the queue increases the access time that is the time to add and remove items from the queue, increases. For this reason and because it is not good practice to have undefined limits on functions in real-time systems queues are rarely used. The circular buffer uses a fixed amount of memory, the size being defined by the designer of the application. If the producer and consumer tasks run normally they would typically add and remove items from the buffer alternately. If for some reason one or the other is suspended for any length of time the buffer will either fill up or empty. The tasks using the buffer have to check, as appropriate, for buffer full and buffer empty conditions and suspend their operations until the empty or full condition changes.

Synchronization with Data transfer:

There are two main forms of synchronization involving data transfer. The first Involves the producer task simply signaling to say that a message has been produced and is waiting to be collected, and the second is to signal that a message is ready and to wait for the consumer task to reach a point where the two tasks can exchange the data. The first method is simply an extension of the mechanism used in the example in the previous section to signal that a channel was empty or full. Instead of signaling these conditions a signal is sent each time a message is placed in the channel. Either a generalized semaphore or signal that counts the number of sends and waits, or a counter, has to be used.

5.11 LIVENESS:

An important property of a multi-tasking real-time system is Liveness. A system (a set of tasks) is said to possess Iiveness if it is free from livelock, deadlock. and indefinite postponement. Livelock is the condition under which the tasks requiring mutually exclusive access to a set of resources both enter busy wait routines but neither can get out of the busy wait because they are waiting for each other. The CPU appears to be doing useful work and hence the term Livelock.

Deadlock is the condition in which a set of tasks are in a state such that it is impossible for any of them to proceed. The CPU is free but there are no tasks that are ready to run.

5.12 MINIMUM OPERATING SYSTEM KERNEL:

As mentioned in the introduction there has been considerable interest in recent years in the idea of providing a minimum kernel of RTOS support mechanisms and constructing the required additional mechanisms for a particular application or group of applications. One possible set of functions and primitives for *RTGS* is:

Functions:

1. A clock interrupts procedure that decrements a time count for relevant tasks,
2. A basic task handling and context switching mechanism that will support the moving of tasks between queues and the formation of task queues.
3. Primitive device routines (including real-time clock support).

Primitives:

WA I T for some condition (including release of exclusive access rights).

S I G N A L condition and thus release one (or all) tasks waiting on the condition,

ACQUIRE exclusive rights to a resource (option - specify a time-out condition).

RELEASE exclusive rights to a resource.

DELAY task for a specified time.

CYCLE task, that is suspend until the end of its specified cyclic period.

Recommended Questions:

1. Draw up a list of functions that you would expect to find in a real-time operating system. Identify the functions which are essential for a real-time system.
2. Discuss the advantages and disadvantages of using
   (a) Fixed table
   (b) Linked list
   Methods for holding task descriptors in a multi-tasking real-time operating system.

3 A range of real-time operating systems are available with different memory allocation strategies. The strategies range from permanently memory-resident tasks with no task swapping to fully dynamic memory allocation. Discuss the advantages and disadvantages of each type of strategy and give examples of applications for which each is most suited.

4. What are the major differences in requirements between a multi-user operating system and a multi-tasking operating system?

5. What is the difference between static and dynamic priorities? Under what circumstances can the use of dynamic priorities be justified?

1. Choosing the basic clock interval (tick) is an important decision in setting up an RTOS. Why is this decision difficult and what factors need to be considered when choosing the clock interval?

2. List the minimum set of operations that you think a real-time operating system kernel needs to support.

8. What is meant by context switching and why it is required?

Module-5

Design of RTSS General Introduction

Introduction, Specification documentation, Preliminary design, Single –Program Approach, Foreground /Background, Multi- Tasking approach, Mutual Exclusion Monitors.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.
3. **Real time Systems Development**, Rob Williams, Elsevier, 2006.

7.1 DESIGN O F RTSS- GENERAL INTODUCTION INTRODUCTION

The approach to the design of real-time computer systems is no different in outline from that required for any computer-based system or indeed most engineering systems. The work can be divided into two main sections:

• The planning phase; and

• The development phase.

It is concerned with interpreting use requirements to produce a detailed specification of the system to be developed and an outline plan of the resources - people, time, equipment, costs - required to carry out the development. At this stage preliminary decisions regarding the division of functions between hardware and software will be made. A preliminary assessment of the type of computer structure - a single central computer, a hierarchical system, or a distributed system - will also be made. The outcome of this stage is a specification or requirements document. (The terminology used in books on software engineering can be confusing; some refer to a specification requirement document as well as to specification document and requirements document. It clearer and simpler to consider that documents produced by the user or customer describe requirements, and documents produced by the supplier or designer give the specifications.) It cannot be emphasized too strongly that the specification document for both the hardware and software which results from this phase must be complete, detailed and unambiguous. General experience has shown that a large proportion of errors which appear in the final system can be traced back to unclear, ambiguous or fault) specification documents. There is always a strong temptation to say 'It can be decided later'; deciding it later can result in the need to change parts of the system which have already been designed. Such changes are costly and frequently lead to the introduction of errors. This shows the distribution of errors and cost of rectifying them (the figures are taken from DeMarco, 1978). The detailed design is usually broken down into two stages:

- Decomposition into modules; and
- Module internal design.

For real-time systems additional heuristics are required, one of which is to divide modules into the following categories:

- Real-time, hard constraint;
- Real-time, soft constraint; and
- Interactive.

The arguments given in Chapter 1 regarding the verification and validation of different types of program suggest a rule that aims to minimize the amount of software that falls into the hard constraint category since this type is the most difficult to design and test.

7.2 SPECIFICATION DOCUMENTATION:

To provide an example for the design procedures being described we shall consider a system comprising several of the hot-air blowers described. It is assumed that the planning phase has been completed and a specification document has been prepared. A PID controller with a sampling interval of 40 ms is to be used. The sampling interval may be changed, but will not be less than 40 ms. The controller parameters are to be expressed to the user in standard analog form, that is proportional gain, integral action time and derivative action time. The set point is to be entered from the keyboard. The controller parameters are to be variable and are to be entered from the keyboard.

7.3 PRELIMINARY DESIGN:

Hardware design:

There are many different possibilities for the hardware structure. Obvious arrangements are:

1. Single computer with multi-channel ADC and DAC boards.
2. Separate general purpose computers on each unit.
3. Separate computer-based microcontrollers on each unit linked to a single general. Purpose computer.

Each of these configurations needs to be analyzed and evaluated. Some points to consider are:

Option 1: given that the specification calls for the system to be able to run with a sample interval for the control loop of 40 ms, can this be met with 12units sharing a single processor?

Option 2: is putting a processor that includes a display and keyboard on each unit an expensive solution? Will communication between processors be required? (Almost certainly the answer to this is yes; operators and managers will not want to have to use separate displays and keyboards.)

Option 3: what sort of communication linkage should be used? A shared high speed bus? A local-area network? Where should the microcontrollers be located? At each blower unit or together in a central location? Each option needs careful analysis and evaluation in terms of cost and performance. The analysis must include consideration of development costs, performance operating and maintenance costs. It should also include consideration of reliability and safety. To provide a basis for consideration of the widest range of approaches to software design we will assume that option 1 above is chosen.

Software design:

Examining the specification shows that the software has to perform several
different functions:

- DDC for temperature control;
- Operator display;
- Operator input;
- Provision of management information;
- System start-up and shut-down; and
- clock/calendar function.

The various functions and type of time constraint are shown in Figure. The control module has a hard constraint in that it must run every 40 ms. In practice this constraint may be relaxed a little to, say, 40 ms ± 1 ms with an average value over 1 minute of, say, 40 ms ± 0.5 ms. In general the sampling time can be specified as *Ts* ± *es* with an average value, over time *T*, of *Ts* ± *ea*. The requirement may also be relaxed to allow, for example, one sample in 100 to be missed. These constraints will form part of the test specification. The clock/calendar module must run every 20 ms in order not to miss a clock pulse. The operator display, as specified, has a hard constraint in that an update interval of 5 seconds is given. Common sense suggests that this is unnecessary and an average time of 5 seconds should be adequate; however, a maximum time would also have to be specified, say 10 seconds.. These would have to be decided upon and agreed with the customer. They should form part of the specification in the requirements document. The start-up module does not have to operate in real time and hence can be considered as a standard interactive module. The sub-problems will have to share a certain amount of information and how this is done and how the next stages of the design proceed will depend upon the general approach to the implementation. There are three possibilities:

- Single program;
- Foreground/background system; and
- Multi-tasking.

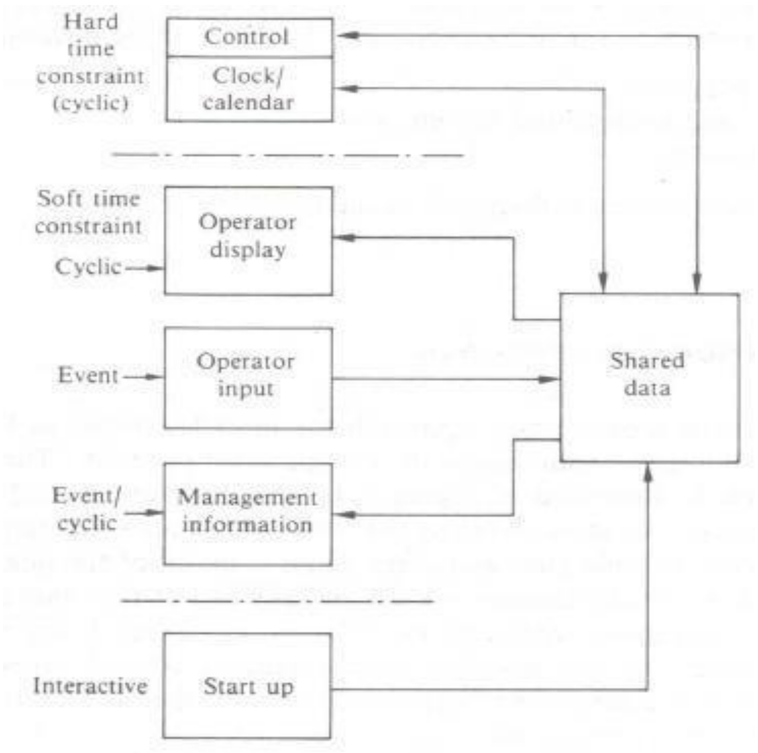


Figure: Basic software modules.

7.4 SINGLE- PROGRAM APPROACH:

Using the standard programming approach the modules shown in Figure are treated as procedures or subroutines of a single main program. The flow chart of such a program is illustrated in Figure. This structure is easy to program; however, it imposes the most severe of the time constraints - the requirement that the clock/calendar module must run every 20 ms - on all of the modules. For the system to work the clock/calendar module and anyone of the other modules must complete their operations within 20 ms. If *fl, fz, f3, f4* and *fs* are the *maximum* computation times for the module's clock/calendar, control, operator display, operator input and management output respectively, then a requirement for the system to work can be expressed as *fl + max (tz, f3, f4, fs)* < 20 ms.

The single-program approach can be used for simple, small systems and it lead to a clear and easily understandable design, with a minimum of both hardware and software. Such systems are usually easy to test.. In the above example the management output requirement makes it unsuitable for the single-program approach; if that requirement is removed the approach could be used. It may, however, require the division of the display update module into three modules: display date and time; display process values; and display controller parameters.

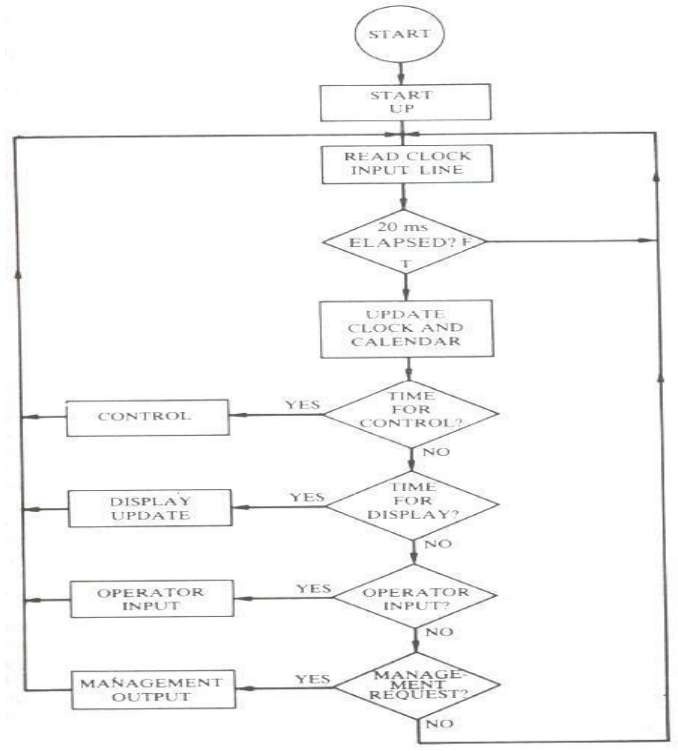


Figure: Single program approach.

7.5 FOREGROUND/BACKGROUND SYSTEMS:

These are obvious advantages - less module interaction, less tight time constraints if the modules with hard time constraints can be separated from, and handled independently of, the modules with soft time constraints or no time constraints. The modules with hard time constraints are run in the so-called 'foreground' and the modules with soft constraints (or no constraints) are run in the 'background'. The foreground modules, or 'tasks' as they are usually termed, have a higher priority than the background tasks and a foreground task must be able to interrupts background task. The partitioning into foreground and background usually requires the support of a real-time operating system, for example the Digital Equipment Corporation's RT/11 system. It is possible, however, to adapt many standard operating systems, for example MS-DOS, to give simple foreground/background operation if the hardware supports interrupts.

The foreground task is written as an interrupt routine and the background task as a standard program. If you use a PC you are in practice using a foreground/background system. The application program that you are using (a word processor, a spreadsheet, graphics package or some program which you have written yourself in a high-level language) is, if we use the terminology given above, running in the background. In the foreground are several interrupt-driven routines - the clock, the keyboard input, the disk controller - and possibly some memory-resident programs which you have installed - a disk caching program or an extended memory manager. The terminology foreground and background can be confusing; literature concerned with non-real-time software uses foreground to refer to the application software and background to refer to interrupt routines that are hidden from the user.

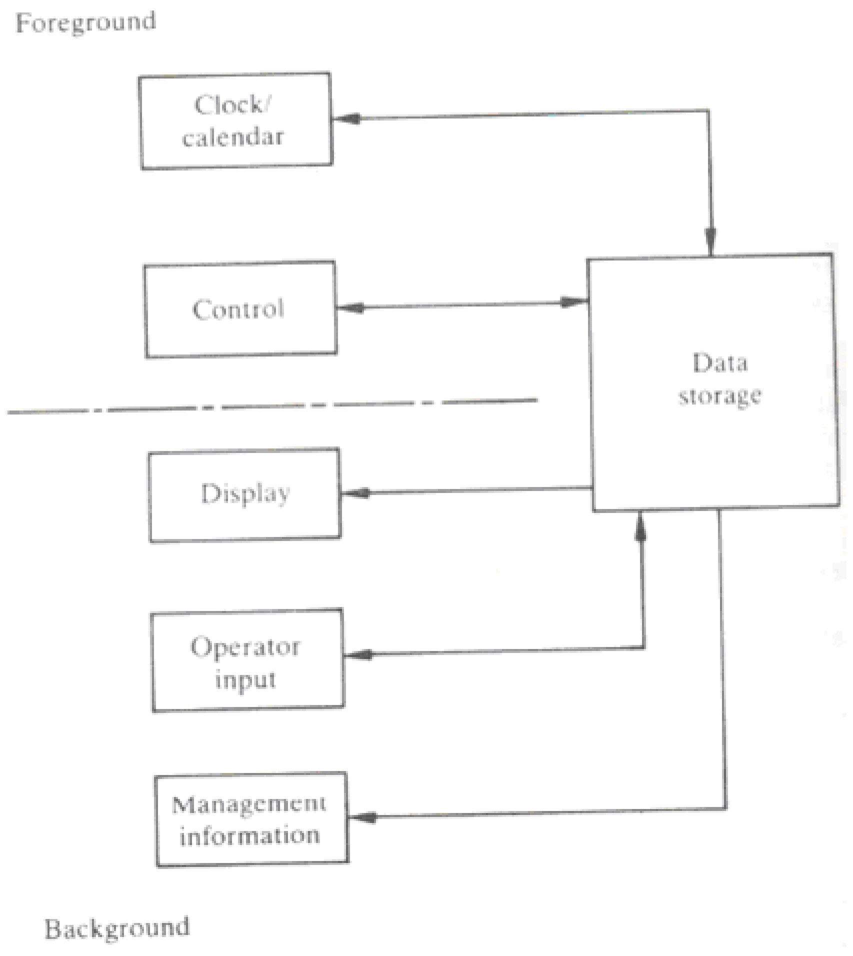


Figure: Software module for foreground/background system.

7.6 MULTI-TASKING APPROACH:

The design and programming of large real-time systems is eased if the foreground/background partitioning can be extended into multiple partitions to allow the concept of many active tasks. At the preliminary design stage each activity is considered to be a separate task. (Computer scientists use the word process rather than task but this usage has not been adopted because of the possible confusion which could arise between internal computer processes and the external processes on the plant.) The implications of this approach are that each task may be carried out in parallel .and there is no assumption made at the preliminary design stage as to how many processors will be used in the system. The implementation of a multi-tasking system requires the ability to:

- Create separate tasks;
- Schedule running of the tasks, usually on a priority basis;

- Share data between tasks;
- Synchronize tasks with each other and with external events;
- Prevent tasks corrupting each other; and
- Control the starting and stopping of tasks.

The facilities to perform the above actions are typically provided by a real-time operating system (RTOS) or a combination of RTOS and a real-time programming language. For simplicity we will assume that we are using only one CPU and that the use of this CPU is time shared between the tasks. We also assume that a number of so-called primitive instructions exist. These are instructions which are part of a programming language or the operating system and their implementation and correctness is guaranteed by the system. All that is of concern to the user is that an accurate description of the syntax and semantics is made available. In practice, with some understanding of the computer system, it should not be difficult to implement the primitive instructions. Underlying the implementation of primitive instructions will be an eventual reliance on the system hardware. For example, in a common memory system some form of arbiter will exist to provide for mutual exclusion in accessing an individual memory location.

7.7 MONITORS:

The basic idea of a monitor is implementation of a monitor in Moouia-2 to protect access to a buffer area is shown. Monitors themselves do not provide a mechanism for synchronizing tasks and hence for this purpose the monitor construct has to be supplemented by allowing, for example, signals to be used within it.

The standard monitor construction outlined above, like the semaphore, does not reflect the priority of the task trying to use a resource; the first task to gain entry can lock out other tasks until it completes. Hence a lower-priority task could hold up a higher-priority. The lack of priority causes difficulties for real-time systems. Traditional operating systems built as monolithic monitors avoided the problem by ensuring that once an operating system call was made (in other words, when a monitor function was invoked) then the call would be completed without interruption from other tasks. The monitor function is treated as a critical section. This does not mean that the whole operation requested was necessarily completed without interruption. For example, a request for access to a printer for output would be accepted and the request queued; once this had been done another task could enter the monitor to request output and either be queued, or receive information from the monitor as to the status of the resource. The return of information is particularly important as it allows the application program to make a decision as to whether to wait for the resource or take some other action.

Preventing lower-priority tasks locking out higher-priority tasks through the monitor access mechanism can be tackled in a number of ways. One solution adopted in some implementations of Modula-2 is to run a monitor with all interrupts locked out; hence a monitor function once invoked runs to completion. In many applications, however, this is too restrictive and some implementations allow the programmer to set a priority level on a monitor such that all lower-priority tasks are locked out - note that this is an interrupt priority level, not a task priority, The monitor has proved to be a popular idea and in practice it provides a good solution to many of the problems of concurrent programming.

The benefits and popularity of the monitor constructs stem from its modularity which means that it can be built and tested separately from other parts of the system, in particular from the tasks which will use it. Once a fully tested monitor is introduced into the system the integrity of the data or resource which it protects is guaranteed and a fault in a task using the monitor cannot corrupt the monitor or the resource which it protects. Although it does rely on the use of signals for inter task synchronization it does have the benefit that the signal operations are hidden within the monitor.

The monitor is an ideal vehicle for creating abstract mechanisms and thus fits in well with the idea of top-down design. However, the nested monitor call problem calling procedures in one monitor from within another monitor can lead to deadlock. Providing that nested monitor calls are prohibited the use of the monitor concept provides a satisfactory solution to many of the problems for a single processor machine or for a multi-processor machine with shared memory, It can also be used on distributed systems.

The monitor's usefulness in some real-time applications is restricted because a task leaving a monitor can only signal and awaken one other task - to do otherwise would breach the requirement that only one task be active within a monitor. This means that a single controlling synchronizer task, for example a clock level scheduler, cannot be built as a monitor. The problem can be avoided by allowing signals to be used outside a monitor but then all the problems associated with signals and semaphores re-emerge.

Recommended Questions:

1. Explain the different phases involved in the design of a RTS.
2. Explain foreground and background system with flowchart.
3. Explain mutual exclusion, using conditional flags.
4. With a neat flow chart, describe the single program approach, with reference to RTS design.
5. Write a note on basic software module, with respect to RTS.
6. Considering a system comprising of several hot air blowers. Prepare specification documents of the same.
7. Explain the data concept of data sharing using common memory.
8. Explain software design for RTS using software module
9. Mention the importance of conditions flag and binary semaphores

RTS Development Methodologies

Introduction, Yourdon Methodology, Requirement definition For Drying Oven, Ward and Mellor Method, Hately and Pirbhai Method.

Recommended book for reading:

1. **Real –Time Computer control –An Introduction**, Stuart Bennet, 2nd Edn. Pearson Education 2005.
2. **Real-Time Systems Design and Analysis**, Phillip. A. Laplante, Second Edition, PHI, 2005.
3. **Real time Systems Development**, Rob Williams, Elsevier, 2006.

8.1 RTS DEVELOPMENT METHODOLOGIES INTRODUCTION

The production of robust, reliable software of high quality for real-time computer control applications is a difficult task which requires the application of engineering methods. During the last ten years increasing emphasis has been placed on formalizing the specification, design and construction of such software, and several methodologies are now extant. All of the methodologies address the problem in three distinct phases. The production of a logical or abstract model - the process of specification; the development of an implementation model for a virtual machine from the logical model - the process of design; and the construction of software for the virtual machine together with the implementation of the virtual machine on a physical system - the process of implementation. These phases, although differently named, correspond to the phases of development generally recognized in software engineering texts. Abstract model: the equivalent of a requirements specification, it is the result of the requirements capture and analysis phase. Implementation model: this is the equivalent of the system design; it is the product of the design stages - architectural design and the detail design

Although there is a logical progression from abstract model to implementation model to implemented software, and although three separate and distinct artifacts abstract model, implementation model, and deliverable system - are produced, the phases overlap in time. The phases overlap because complex systems are best handled by a hierarchical approach: determination of the detail of the lower levels in the hierarchy of the logical model must be based on knowledge of higher - level design decisions, and similarly the lower-level design decisions must be based on the higher-level implementation decisions. Another way of expressing this is to say that the higher-level design decisions determine the requirements specification for the lower levels in the system.

8.2 YOURDON METHODOLOGY:

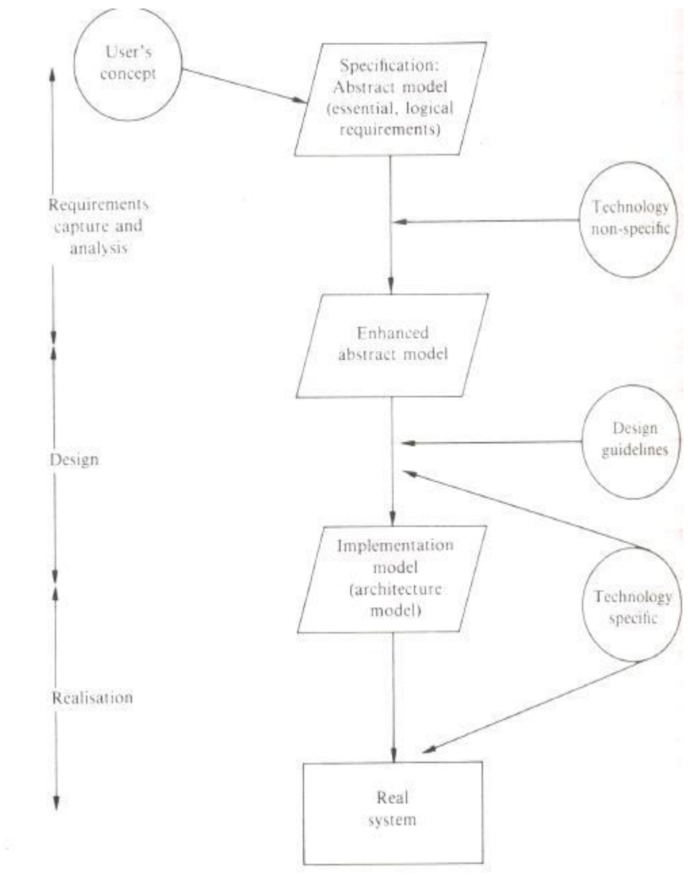

User's concept
Specification: Abstract model (essential, logical requirements)
Requirements capture and analysis
Technology non-specific
Enhanced abstract model
Design
Design guidelines
Implementation model (architecture model)
Technology specific
Realisation
Real system

Figure: Software modeling

The Yourdon methodology has been developed over many years. It is a structured methodology based on using data-flow modeling techniques and junctional decomposition. It supports development from the initial analysis stage through to implementation. Both Ward and Mellor (1986) and Hatley and Pirbhai (1988) have introduced extensions to support the use of the Yourdon approach for the development of real-time systems and the key ideas of their methodologies are:

- Subdivision of system into activities;
- Hierarchical structure;
- Separation of data and control flows;
- No early commitment to a particular technology; and
- Traceability between specification, design and implementation.

8.3 REQUIREMENT DEFINITION FOR DRYING OVEN:

Components are dried by being passed through an oven. The components are placed on a conveyor belt which conveys them slowly through the drying oven. The oven is heated by three gas-fired burners placed at intervals along the oven. The temperature in each of the areas heated by the burners is monitored and controlled. An operator console unit enables the operator to monitor and control the operation of the unit. The system is presently controlled by a hard wired control system. The requirement is to replace this hard wired control system with a computer-based system. The new computer-based system is also to provide links with the management computer over a communication link.

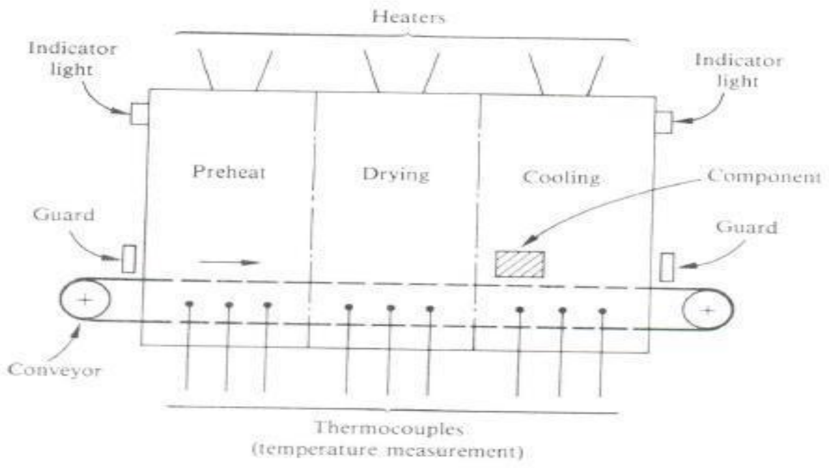


Figure: General arrangement of drying oven.

I
nput/output

The inputs come from a plant interface cubicle and from the operator. There will need to be inputs obtained from the communication interface.

Plant Inputs

A thermocouple is provided in each heater area - the heater areas are pre-heat, drying, and cooling. The inputs are available as voltages in the range 0 to 10 volts at pins 1 to 9 on socket j22 in the interface cubicle.

The conveyor speed is measured by a pulse counting system and is available on pin 3 at socket j23 in the interface cubicle. It is referred to as con-speed..

There are three interlocked safety guards on the conveyor system and these are in-guard, out-guard, and drop-guard. Signals from these guards are provided on pins 4, 5, 6 of socket j23. These signals are set at logic HIGH to indicate that the guards are locked in place.

A conveyor-halted signal is provided on pin I of socket j23. This signal is logic HIGH when the conveyor is running.

Plant Outputs

Heater Control: each of the three heaters has a control unit. The input to the control unit is a voltage in the range 0 to 10 volts which corresponds to no heat output to maximum heat output. Conveyor Start-up: a signal convey-start is output to the conveyor motor control unit.

Guard Locks: asserting the *guard-lock* line, pin 8 on j10 , causes the guards to be locked in position and turns on the red indicator light on the outside of the unit. Operator Inputs

The operator sends the following command inputs: *Start, Stop, Reset, Re-start,* and *Pause.* The operator can also adjust the desired set point for each area of the dryer. Operator Outputs

The operator VDU displays the temperature in each area, the conveyor belt speed, and the alarm status. It should also display the current date and time and the last operator command issued.

8.4 WARD AND MELLOR METHOD:

The outline of the Ward and Mellor method is shown in Figure. The starring point is to build, from the analysis of the requirements, a software model representing the requiremel1ls in terms of the abstract entities. This model is called the essential model. It is in two parts: an environmental model which describes the relationship of the system being modeled with its environment; and the behavioral model which describes the internal structure of the system.

The second stage the design stage - is to derive from the essential model an implementation model which defines how the system is implemented on a particular technology and shows the allocation of parts of the system to processors, the subdivision of activities allocated to each processor into tasks, and the structure of the code for each task. The essential model represents what the system is required to do; the implementation model shows how the system will do what has to be done. The implementation model provides the design from which the implementers of the physical system can work. Correct use of the method results in documentation that provides traceability from the physical system to the abstract speci- fication model.

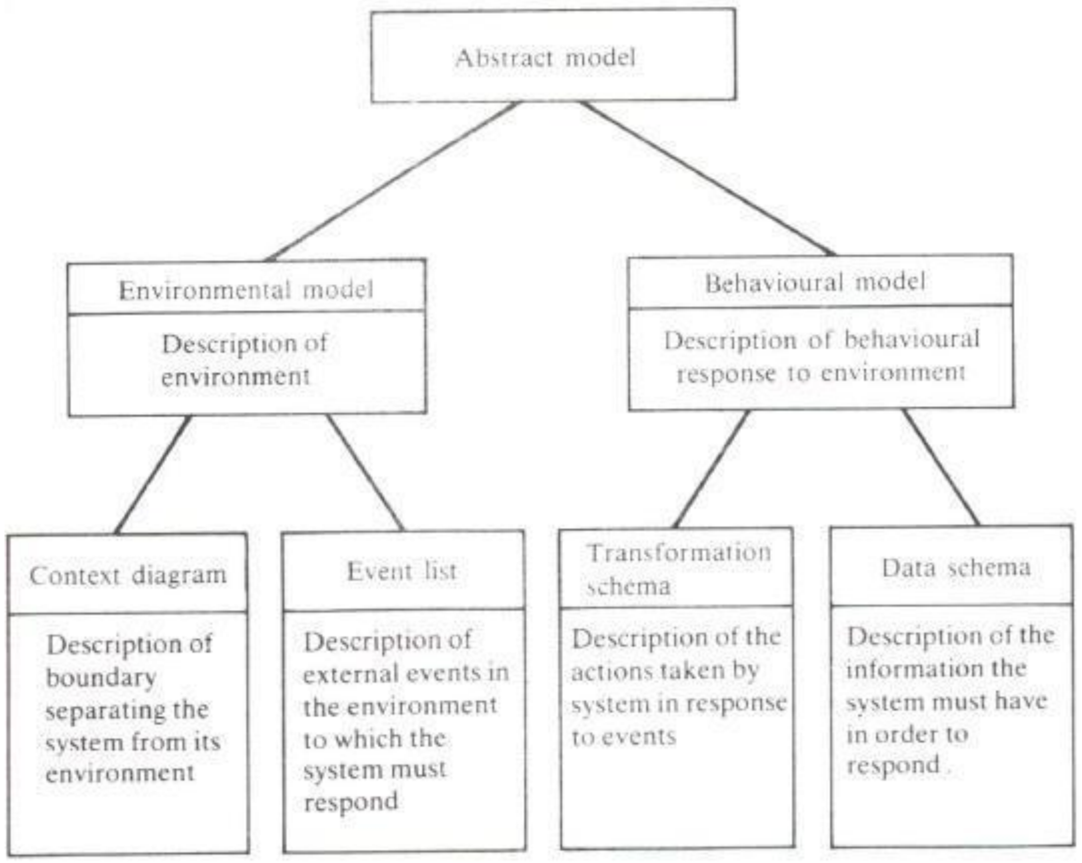
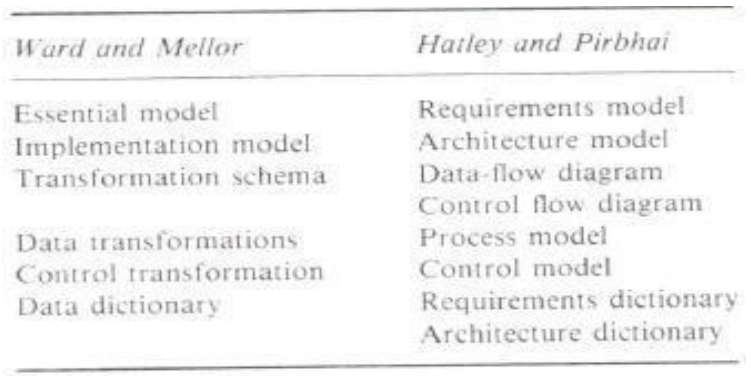

Abstract model
Environmental model
Description of environment
Behavioural model
Description of behavioural response to environment
Context diagram
Description of boundary separating the system from its environment
Event list
Description of external events in the environment to which the system must respond
Transformation schema
Description of the actions taken by system in response to events
Data schema
Description of the information the system must have in order to respond

Figure: Outline of abstract of modeling approach Ward and Mellor.

8.5 HATELY AND PIRBHAI METHOD:

As might be expected the general approach of the Hatley and Pirbhai methodology is very close to that of Ward and Mellor. There are some differences in terminology which are summarized in Table.



Ward and Mellor	*Hatley and Pirbhai*
Essential model	Requirements model
Implementation model	Architecture model
Transformation schema	Data-flow diagram
	Control flow diagram
Data transformations	Process model
Control transformation	Control model
Data dictionary	Requirements dictionary
	Architecture dictionary

Separate diagrams are used for data and control;

- only one CSPEC can appear at any given CFD level; and
- all data f10ws and control f10ws are shown with single arrow heads;

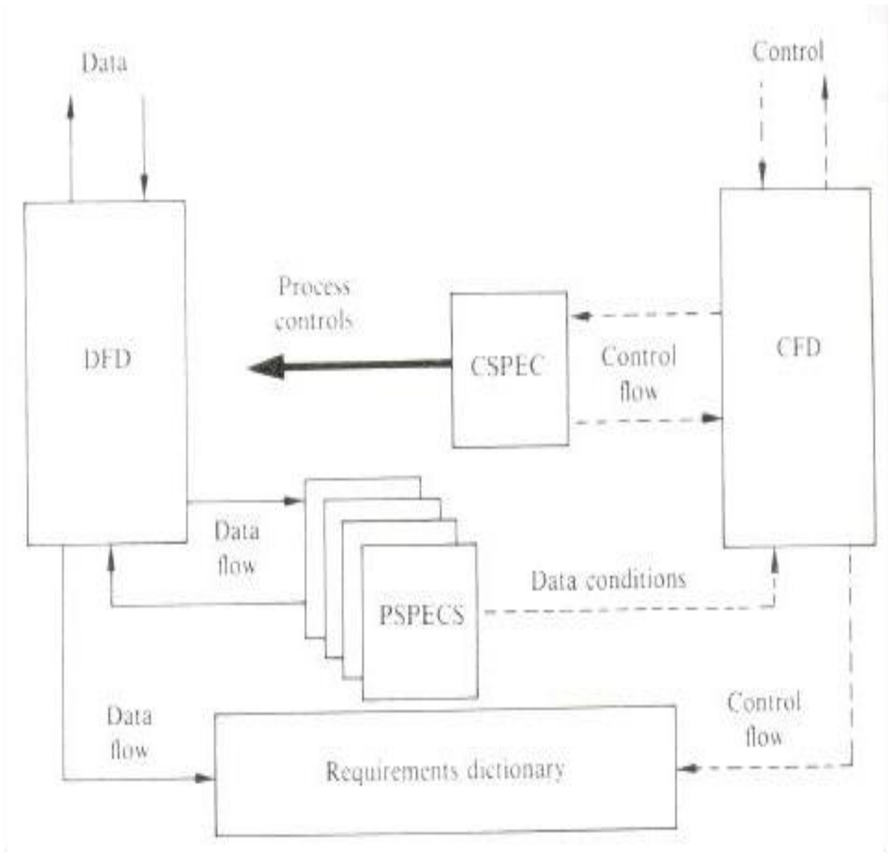
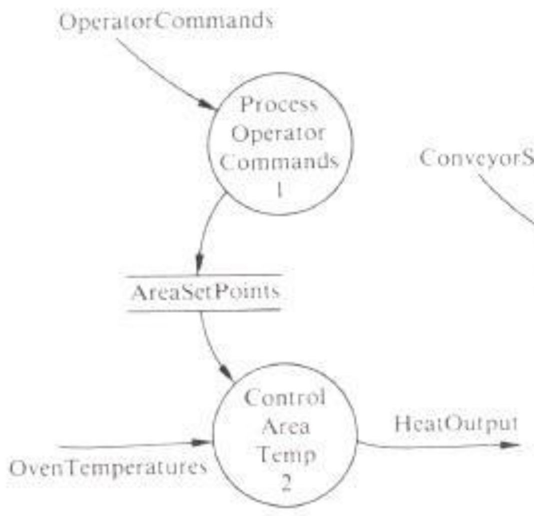
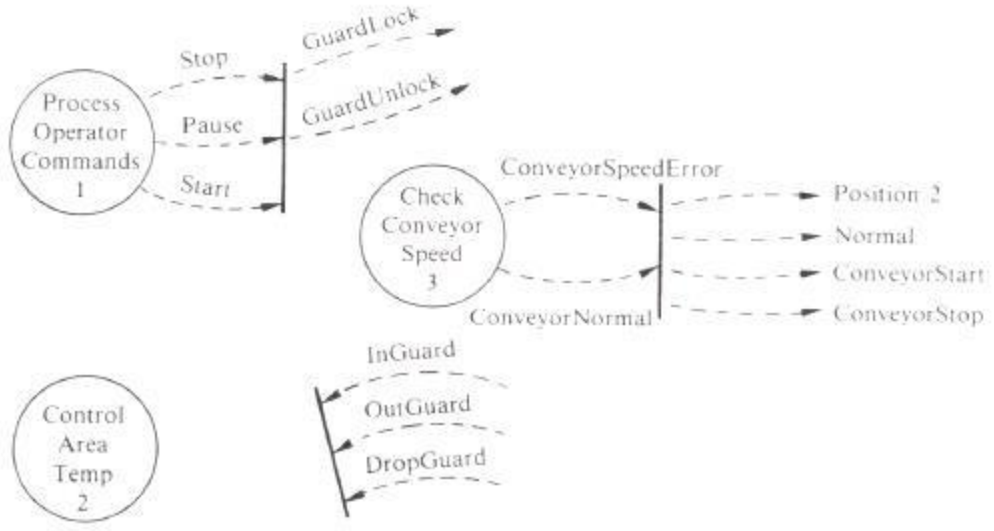


Figure: Structure of requirement model.



DFD 0 Drying Oven Controller



CFD 0 Drying Oven Controller

Figure: Hately and Pirbhai notation.

8.6 COMMENTS ON THE YOURDON METHODOLOGY:

Both methodologies - Ward and Mellor and Hatley and Pirbhai - are simple to learn and have been widely used. They are founded on the well-established structured methods developed by the Yourdon organization and hence over the years a lot of experience in using the techniques has been gained. For serious use on large scale systems they both require the support of CASE tools. The labour involved in checking the models by hand is such that short cuts are likely to be taken and mistakes are bound to occur. It can be argued that the methods are really only a set of procedures for documenting a specification and a design and to some extent this is true.

The analysis procedures are minimal and adequate checking for consistency can be performed only with the support of a CASE tool. However, the methodologies are still useful in that the procedures they recommend provide a sensible way of preparing both a specification and a design in that they encourage the development of hierarchical, modular structures. the two, the Hatley and Pirbhai method is the more structured and formalized in its approach. Its diagrams are less cluttered than those of the Ward and Mellor method and, once the separation is understood, are easier to follow. Many CASE tools provide alternative displays which allow a choice of either separate diagrams or a combined diagram with switching between the two forms. The weakness of both methods lies in the allocation of processors and tasks. The suggestion that one allocates activities to processors and then subdivides the activities into tasks allocated to each processor appears at first sight a sensible way to proceed. However, when it is tried one soon realizes that the information required to do this is not available.

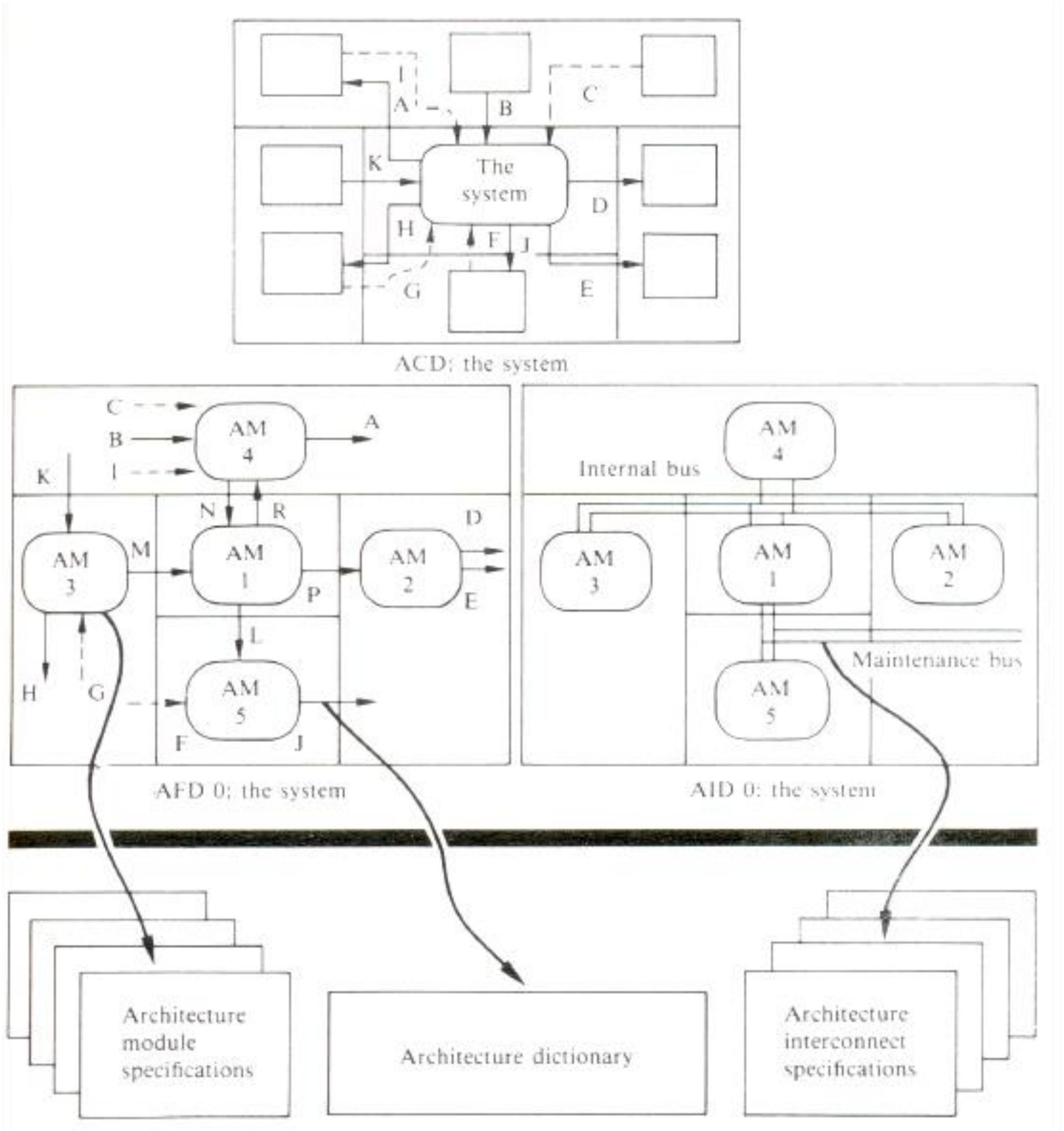


Figure: The structure of architectural model.

Recommended Questions:

1. With a general arrangement for a drying oven, explain its requirements.

2. Write about environmental model, with context diagram for drying oven.

3. Write explanatory notes on the following: A).Hatley and pirbhai methods. B).Ward and millar methods.

4. Show the outline of abstract modeling approach of ward and Mellor and explain.
5. Differentiate between Ward Mellor and hatley and pirbhai mythologies.
4. Explain the CFDO drying over controller using Hatley and pirbhai notation.

6. What do you mean by enhancing the model? Explain with a neat diagram, the relationship between real environment and virtual environment.
7. Write short notes on: i) PSPECs and CSPECs ii) Software modeling iii) YOUR DON methodology