1. INTRODUCTION

The computer system may operate *sequentially* performing tasks handling inputs, outputs, processing functions and communication being carried out between these tasks in turn with the sequence then being repeated indefinitely; or it may operate in a parallel manner, with the various tasks being carried out *concurrently.* In the case of a computer system with a single processing unit full parallel operation cannot be achieved and the system is said to operate *pseudo-concurrently.* The use of concurrency (or pseudo-concurrency) gives rise to problems relating to synchronising the various tasks and the sharing of resources between these tasks

1.1. Fundamentals

1.1.1. Definition of Real-Time Systems

Real-time systems are any systems in which the time at which their outputs are produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The time interval from input time to output time must be, sufficiently small for acceptable timeliness. Real-time systems are those which must produce correct responses within a definite time limit. Should computer responses exceed these time bounds then performance degradation and/or malfunction results. They read inputs from their environment and send output signals to their environment at times determined by operational considerations of their environment- not at times limited by the capabilities of the computer system.

Real-time systems are characterized by the fact that severe consequences will result if logical as well as timing properties of the system are not satisfied. Typically, a real-time system consists of a controlling system and a controlled system. For example, in an factory, the controlled system is the factory floor with its robots, assembling stations and the assembled parts, while the controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor. Thus, the controlled system can be viewed as the environment with which the computer interacts. The controlling system interacts with its environment based on the information available about the environment, say, from various sensors attached to it. It is imperative that the state of the environment, as perceived by the controlling system, be consistent with the actual state of the environment. Otherwise, the effects of the controlling systems' activities may be disastrous. Hence, periodic monitoring of the environment as well as timely processing of the sensed information is necessary.

Timing requirements and their correctness in a real-time system also arise because of the physical impact of the controlling systems' activities upon its environment. For example, if the computer controlling a robot does not command it to stop or turn on time, the robot might collide with another object on the factory floor. Such a mishap can result in a major catastrophe.

Real-time systems cover a very wide range of application areas. Applications can be found in avionics, undersea exploration, process control, robot and vision systems, as well as in military applications, computer networks and systems. There are applications 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. The complexity of real-time systems also spans the gamut from very simple control of laboratory experiments, to process control applications, to very complicated projects such as the space station.

In most of these systems, activities that have to occur in a timely fashion coexist with those that are not time-critical. There is an example of an accounting application - not time-critical - and a production process running on a computer - time-critical - in the same factory. Both types of these activities are denoted as tasks. If a task has a time requirement then this task is *a real-time* task.

Real-time tasks are sometimes called *time-critical* tasks. Ideally, the computer should execute time-critical tasks so that each task will meet its time requirement, whereas it should execute the non-time-critical tasks so that the *average* response time of these tasks is minimised. The need to meet the requirements *of individual* time-critical tasks is an issue that makes the problem of designing a real-time system a hard problem. Other issues include fault tolerance and the need to operate in uncertain environments.

This brings us to the question of what happens when timing constraints are not met. The answer depends, for the most part, on the type of application. For example a real-time system that controls a nuclear power plant, or one that controls a missile, cannot afford to miss timing constraints. Resources needed for highly critical tasks in such systems have to be pre-allocated so that the time-critical tasks can be executed without delay. In many situations, however, some delay does exist. For example, even on a factory, if it is estimated that the correct command to a robot cannot be generated on time, it may be appropriate to command he robot to stop (provided it will not cause other moving objects to collide with it and result in a different disaster). This is an instance of a real-time task introducing a result of lower quality, but on time. In be case of a periodic task monitoring an aircraft, depending on the aircraft's trajectory, missing the processing of one or two radar readings may not cause problems.

In summary, real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to tasks, the systems are in a position to make compromises, and faults may cause catastrophic consequences. This implies that, unlike many systems where there is a separation between correctness and performance, in real-time systems correctness and performance are very tightly interrelated. Thus, rea1-time systems solve the problem of missing deadlines in ways specific to the requirements of the target application. However, it could be said that the sooner a system determines that deadline is going to be missed, the more flexibility it will have in dealing with the exception.

Because timing properties of real-time systems, specifically the execution times of the various tasks, every tightly related to the underlying hardware, quite often an early binding of the logical functions and the physical hardware units is done. The point at which this binding is done, in part, determines the adaptability and predictability properties of the system.

Clearly, depending on the complexity of a controlled system, the static versus dynamic nature of the activities of the environment, the complexity of the tasks executed by the controlling system, and the overall goal of the system, design and implementation strategies adopted will vary.

1.1.2. Constraints of Real-Time Systems

It is now common practice to divide real-time systems (and real-time tasks) into two categories:

- *hard real-time systems:* these are systems that must satisfy the deadlines on each and every occasion. An-example of a hard real-time system is the temperature control system. In control terms, the temperature loop is a sampled data system. Design of a suitable contro1 algorithm of this system. involves the choice of the sampling interval Ts. During this interval the input value must be measured and output value calculated and sent to the actuator.
- *soft real-time systems:* these are systems for which an occasional failure to meet a deadline does not comprise the correctness of the system. 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 respond will be specified in terms of an average response time . The actual response time will vary; if you have used such a system you will have learned that the response time obtain between 12 and 2 p.m. on a Friday is very different from that at 10 a.m. on Sunday). An automatic bank teller would not be satisfactory if at busy times customers had wait 10 minutes, even if it achieved a mean response measured over a 24 hour period of 20 seconds

Timing constraints of real-time systems can be formally defined as follows:

1. hard time constraints

periodic (cyclic) $t_c(i) = t_s \pm a$

aperiodic (event) $t_e(i) << T_e$

2. soft time constraints

periodic (cyclic) $$\frac{1}{n}\sum_{i=1}^{n} t_c(i) = t_s \qquad n = \frac{T}{t_s} \tag{1.1}$$

aperiodic (event) $$\frac{1}{n}\sum_{i=1}^{n} t_e(i) \le T_a \qquad n = \frac{T}{t_s} \tag{1.2}$$

where

t_c (I) - the interval between the "i"and i -1 cycles,

$t_e(i)$ - the response time to the ith occurrence of event "e",

t_s - the desired periodic (cyclic) interval,

T_e - the maximum permitted response time to event "e",

T_a - 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 - timing tolerance.

A hard time constraint obviously represents a much more severe constraint to the performance of the system than a soft time constrains and such systems present a difficult challenge both to hardware engineers 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.

Timing constraints for tasks can be arbitrarily complicated, but the most common timing constraints for tasks are either periodic or aperiodic. An aperiodic task has a deadline by which it must finish or start, or it may have time constraints to both start and finish times. In the case of a periodic task, a period is a time interval during that the periodic task must be performed. Low-level application tasks, such as those that process information obtained from sensors, or those that activate elements in their environment, typically have stringent timing constraints dictated by the physical characteristics of the environment. A majority of acquisition input data from the environment and its processing is periodic in nature. For example, a radar that tracks flights produces data at a fixed rate. A temperature monitor of a nuclear reactor should be read periodically to detect any changes. Some of these periodic tasks may exist from the point of system initialisation, while others may come into existence dynamically. The temperature monitor above is an instance of a permanent task. An example of a dynamically created periodic task is a task that monitors a particular flight. It comes into existence when the aircraft enters an air traffic control of a region and will cease to exist when the aircraft leaves the region.

More complex timing may also occur. For example, spray painting a car on a moving conveyor must be started after time t_1 and completed before time t_2. Aperiodic requirements can arise from dynamic events such as an object falling in front of a moving robot or an operator pushing a button on a console. In addition, time-related requirements may also be specified in indirect terms. For example, a value may be attached to the completion of each task where the value may increase or decrease

with time; or a value may be placed on the quality of an answer whereby an inexact but fast answer might be considered more valuable than a slow but accurate answer. In other situations, missing one group of deadlines might be tolerated, but missing another group of deadlines cannot be tolerated.

For some systems and tasks the timing constraints may be combined in some form or other, or relaxed in some way. For example, soft aperiodic (event) time constraints are often combined with a hard time constraint. In some circumstances an occasional missed periodic hard time constraint might not be serious; for example, in a feedback control system missing the occasional sample will not result in a major disturbance of the controlled system.

Characteristic of the various application are usually known a priori and tasks performing them may be scheduled statically or dynamically. Whereas static specification of schedules is typically the case for periodic tasks, the opposite is true for aperiodic tasks. When the temperature monitor of our previous nuclear reactor example senses a problem in the core, it can invoke another (aperiodic) task to activate the appropriate elements of the reactor to correct the problem, for example, to force more coolant into the reactor core. In this case, the deadline for the aperiodic task can be statically set depending on the physical characteristics of the reasons within the core. On the other hand, the deadline of a task that controls a robot on a factory floor can be determined dynamically depending on the speed, direction, etc., of the robot. The command to the robot forcing it to turn right, left, or stop should be generated before this time.

Because timing properties of real-time systems, specifically the execution times of the various tasks, every tightly related to the underlying hardware, quite often an early binding of the logical functions and the physical hardware units is done. The point at which this binding is done, in part, determines the adaptability and predictability properties of the system. Clearly, depending on the complexity of a controlled system, the static versus dynamic nature of the activities of the environment, the complexity of the tasks executed by the controlling system, and the overall goal of the system, design and implementation strategies adopted will vary.

In addition to time constraints, a task may also posses the following types of constraints and requirements :

- *resource constraints:* A task may require access to certain resources other than the CPU such as I/O devices, data structures, files, and databases.
- *precedence constraints:* A complex task, for example, one requiring access to many resources, may be handled by decomposing it up into multiple subtasks taking into account precedence constraints. Each subtask will require a subset of resources.
- *concurrency constraints;* Tasks should be allowed concurrent access to resources providing the consistency of the resources is not violated their timing requirements: Sets of cooperating tasks will be the norm for distributed real-time systems. The semantics of the concurrent tasks will vary as well as the interconnection structure between the communicating tasks and their timing requirements.
- *placement constraints:* When multiple instances of a task are executed in order to achieve fault-tolerance, the different instances should be executed on different processors.
- *criticalness:* Depending on the functionality of a task, meeting the deadline of one task may be considered more critical than another. For example, a task that reacts to an emergency situation such as fire on the factory floor, in all likelihood, will be more critical than the task that controls the movements of a robot under normal operating conditions.

1.1.3. Properties of Real-Time Systems

In a system that is static, the characteristics of the controlled system are assumed to be known a priori, and, hence, the nature of activities and the sequence in which these activities take place can be determined before the system begins operation. Needless to say, such systems are quite inflexible

even though they may incur lower run-time overheads. In practice, most applications involve a number of components that can be statically tackled along with many dynamic components. If handled appropriately, a system with high resource utilisation and low overheads can be produced for such applications. A large proportion of currently implemented real-time systems are static in nature. Next generation systems will have to adopt solutions that are more dynamic and flexible. This is because such systems will be large and complex and they will function in environments that are dynamic while being physically distributed. More important, they will have to be maintainable and extensible due to their evolving nature and proposed long lifetimes. Real-time systems in general, and systems with these characteristics particular, need to be first predictable, reliable and adaptive.

Predictability A system has to be both fast and predictable. Predictability means that when a task is activated it should be possible to determine its completion time with certainty. This must be done taking into account the state of the system (including the state of the operating system and the state of the resources controlled by the operating system) and the task's resource needs. For a predictability of a system the following questions must be answered :

- Will task meet its timing requirements
- How many tasks will finish before their deadline?
- What will be the maximum utilisation of a given resource?
- When will a task finish?

Reliability is a prerequisite for real-time systems. Real-time constraints cannot be achieved if system components are not reliable. Level of reliability should be specificable and predictability will involve determining system performance under different levels of reliability.

Adaptability A real-time system should be adaptive to changes in system state (i.e., system dynamics including overloads and failures (short-term)) to changes in system configuration (long-term) to changes in input specifications; in particular, task specifications (long-term). Adaptability is particularly important for real-time systems because if a task's deadlines can be met only under a restricted system state/configuration, reliability and performance may be compromised. If a system is adaptive, one does not have to redefine the system or recompute resource and task allocation for every small change. This reduces development and maintenance costs. To be predictable and adaptive at the same time we need "carefully designed" schemes. A real-time system should be adaptive to changes in system state (i.e., system dynamics including overloads and failures (short-term)) to changes in system configuration (long-term) to changes in input specifications; in particular, task specifications (long-term).

1.1.4. State of Art in Real-Time Systems

Specification and verification. Existing practices for designing, validating, and implementing real-time systems are rather ad hoc today. Software engineering uses modularity and abstract data types in non-real-time systems. In contrasts this is not the case in real-time software design as a result of real-time requirements.

The problem of specification, design, and analysis of real-time programs is made difficult by the parallelism inherent in real-time applications and is exacerbated further by the presence of hard real-time constraints. Design and application engineers are interested not only in the logical properties of safety and lifeless but also in the physical property of timeliness.

The specification for real-time systems is a kind of contract between the system and the environment. Based on certain assumed properties of the environment, the system guarantees to provide the specified performance. The design of the system should not only meet the specifications but also guard the system against faults in the behaviour of the environment (i.e., against deviations from its assumed behaviour). A .number of proposals, based on extensions to existing formalisms for concurrent systems, have been made to deal with the specification and analysis of real-time systems. While there is ongoing work on the use of temporal logic for specifying and analysing real-time

systems, the majority of current work is based on finite state machines, Petri nets, CSP logic, and first order logic. New specification and verification techniques are required to provide answers to the following questions:

- What is the worst-case response time of the system to a particular type of external event? Is the system performance under peak arrival rates adequate?
- What· is the effect of different operating system policies, such as scheduling policies, on safety and timeliness?
- Given the specification of a system, do the specifications conform to the safety properties of interest to the system?
- Given an implementation, how can one ensure that the implementation conforms to the specifications?
- What are the restrictions imposed on the external environment for the system to react safely?
- What are the rates at which stimuli can arrive at the system?
- What will be the effect of failures in the external environment?

Real-time operating systems. It is often the case that existing real-time systems are supported by stripped-down and optimised versions of time-sharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system fast, the kernel underlying a current real-time system has a fast context switch that has a small size with associated minimal functionality. It is provided with the ability to respond to external interrupts quickly to minimise intervals during which interrupts are disabled. It also provides fixed or variable sized partitions for memory management, i.e., no virtual memory as well as the ability to lock code and data in memory. It has special sequential files that can accumulate data at a fast rate. To deal with timing requirements, the kernel maintains a real-time clock. The kernel provides special alarms and time-outs tasks that can invoke primitives to delay by a fixed amount of time and to pass and/or resume execution. In general, the kernels perform multi-linked inter-task communication and synchronisation that are achieved via mailboxes, proxies, events and semaphores. These features are also designed to be fast. However, fast is a relative term and not sufficient when dealing with the real-time constraints. Nevertheless, these features can be said to provide a basis for a good set of primitives upon which to build real-time systems.

The key issue that has emerged is the need to provide predictability. Predictability requires clean operating system primitives, some knowledge of the application, proper scheduling algorithms, and a viewpoint based on a team attitude between the operating system and the application. The operating system must be able to perform integrated CPU scheduling and resource allocation so that collections of cooperating tasks can obtain the resources they need, at the right time, in order to meet timing constraints. The cooperation requirement means that there is an end-to-end timing requirement, i.e. a collection of activities must occur possibly with complicated precedence constraints before some deadline. Using the current operating system paradigm of allowing arbitrary waits for resources or events, or treating the operation of a task as a random process, will probably not be feasible in the future to solve this more complicated set of requirements. An important realistic and complicating factor to the integrated resource allocation problem is the need to be predictable in the presence of faults. This issue must receive more attention. It is also important to avoid rewriting the operating system for each application area. A library of real-time operating system objects might provide the Bevel of portability required. It could be similar to a Smalltalk-like system for hard real time constraints, so that a designer can tailor the OS to the application without having to write everything from scratch. Unfortunately, there is no such system available, although some work is going on this approach.

Scheduling The problem with current technology, however, lies in priority driven or table-driven schedulers , i.e. how they handle tasks with real-time constraints. The primary considerations in task

scheduling today are the execution times of tasks taking into consideration time constraints of tasks. Task execution times are, in general, stochastic: The worst-case execution time of a task may be much higher than its average execution time. Whereas the timing constraints are an indication of the level of urgency of a real-time task, the criticalness of a task is indicative of the level of importance that is attached to that task relative to the other tasks. Providing enough processing power so that even at peak loads (i.e., worst-case situations) all tasks will meet their timing requirements then it will result in a highly under-utilised system. However, real-time systems such as those used in air traffic control and military command and control are called upon to perform efficiently and correctly especially in overloaded situations. The solution lies in scheduling tasks such that the highly critical tasks are always guarantied to meet their timing constraints. However, because there is normally no correlation between task deadlines and criticalness, scheduling tasks so as to maximise the number of highly critical tasks that meet their deadlines is a non-trivial problem.

Today's real-time systems continue to use priority-driven schedulers for scheduling tasks with timing constraints. This is a carry-over from the approach taken in kernels of time-sharing systems. This implies that the two competing considerations, namely, time constraints. and criticalness values of tasks, are to be mapped into one factor, namely, the priority of a task. Typically, this is done through an iterative process involving extensive simulation studies: Initially, task priorities reflect just their criticalness values and the system is tested. Task priorities are adjusted if some critical tasks miss their deadlines or if the resource utilisation is poor. Optimisation of task code is often performed. The changes to task code as well as task priorities continues until a satisfactory performance is achieved in conjunction with priority based scheduling, many of today's real-time systems also use a small set of executives each executing at a different frequency. Tasks in a system execute in either the major cycle (the cycle with the highest frequency) or in one of the minor cycle (the cycle of lower frequency). The major cycle is assigned the highest priority, and, hence, the highest frequency periodic tasks as well as the highly critical (but lower frequency) tasks are associated with the major cycle. The order in which the tasks in the various cycles should be executed is specified in a table whose construction requires a number of iterations involving the adjustment of the frequency of the cycles, modifications to the ordering of tasks within the table , and task code optimisation.

In summary, today's priority-based scheduling technology involves a lot of code and priority tuning accompanied by extensive simulations to gain confidence in the system. Even though these scheduling techniques have been successfully used in many real-time embedded systems, it is only at extremely high cost and inflexibility. Because the scheduling technique is not algorithmical, it is error-prone. For example, it is difficult to predict how tasks invoked dynamically interact with other active tasks, where blocking over resources will occur, and the subsequent effect of this interaction and blocking on the timing constraints of all the tasks. In addition, even a small change is subject to another extensive round of testing. Overall, this "design philosophy" forces designers and engineers to deviate from structured system development techniques whereby the resulting system is neither adaptive nor extensible. This makes maintenance and upgrading very expensive. But, since typical embedded systems, such as automated factories and air traffic control systems, have long lifetimes during which the systems evolve, incremental development and dynamic maintainability are of importance to these systems.

Real-Time Languages. There are certain minimal requirements that should be satisfied by any language used for building real-time systems. These languages should be high-level so that a user need not be very intimately involved with the underlying hardware architecture. This also allows the use of many formal analysis techniques and tools that are enable only to high-level programs. Modules should be programmable such that they have (a priori determinable) bounded execution times. Hence, the use of dynamic structures and unbounded loops should be disallowed or carefully conceded. Since predictability is one of the primary characteristics of real-time systems, all program constructs and modules should be analysable with respect to the schedulability of modules, availability of resources, and satisfaction of system-timing constraints. The last point, analysability of programs written in a real-time programming language, will be an important criterion choosing a

language for constructing real-time systems of the future. However, in the absence of good, automatic analysis techniques, a lower-level view of programming languages for real-time *systems* prevails. By low level we mean that the programmer explicitly controls parallelism, priority, and timing.

Hardware and Structures Hard real time systems are usually have special purpose hardware. Architectures that support such applications also tend to be special purpose ones. However, a number of general concepts and principles have emerged from various architectural and hardware developments. Unfortunately not all of them could be applied in all real-time systems. As a result there is a need to develop special purpose off-the-shell hardware modules of real-time computers that do not change with each application. Real-time computers must provide hardware fault tolerance and they must satisfy real-time constraints. The capacity and size of these systems are strongly influenced by the growth of overhead as modules are added. Program code of real-time computers might have to be in ROM to withstand harsh environments. There is a need to perform functional partitioning of program code, but too rigid partitioning and static schedule would be not suitable in many systems. Real-time computers must also include on-line testability. To decrease system overhead of processors and to keep application tasks free local memories can be used for read-only code. Global memory can contain shared data because dynamically loaded local memories and caches cause many problems difficult to deal with under timing constraints because they may violate a principle that repeatability of timings is crucial.

Communication in Real-Time Systems: In the design of distributed real-time systems, there is a need for the use of communication protocols that provide deterministic behaviour of the communicating components. In particular, this requires protocols that result in bounded message communication delays. A number of advances have been made with regard to protocols for time-constrained communication. New access control protocols that are more efficient than pure TDMA are being developed. At the same time, new collision-based protocols that are more predictable than pure CSMA protocols continue to emerge. However, integration of the low-level protocols with the functioning of the operating system kernel, I/O modules, and application modules, as well as the inclusion of fault-tolerant features, must be solved.

Fault Tolerance: Fault tolerance must be designed into real-time systems. It must encompass both hardware and software fault tolerance and it must be integrated with timing constraints. In many cases, the fault-tolerant design must be static as a result of extremely high data rates and severe constraints. Highly-reliable systems need to employ proof-of-correctness techniques to ensure fault tolerance properties. Primary and back-up schedules computed off line are often found in hard real-time systems to provide fault tolerance. In new approaches on-line schedulers predict that timing constraints will be missed, enabling early action on such faults. Dynamic reconfigurability is also needed, but little progress has been reported in this area. Also, while considerable advance has been made in the area of some fault tolerance techniques that explicitly take into account timing of application fault tolerance in real-time systems are lacking.

1.2. Classification of Real-Time Systems

A common feature of real-time systems is that the computer is connected to the environment within which it is working by a wide range of interface devices and receives and sends a variety of stimuli. For example, the plant input, plant output, and communication tasks have one feature in common - they are connected by physical devices to processes which are external to the computer. These external processes all operate in their own time scales 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.

Synchronisation between the external processes and the internal actions (tasks) carried out by the computer may be defined in terms of the relative time, or the actual time of day, in which case the system is said to be *clock-based* and the operations are carried out according to a time schedule. Or it may be defined in terms of events in which case it is said to be *event-based.* There is a third category,

interactive, in which the relationship between the actions in the computer and the system is much more loosely defined. Typically, the requirement is that a set of operations in the computer should be completed within a predetermined time. The majority of communication tasks fall into this category.

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 used. It is useful to divide tasks to be carried out by computers into the above categories and users should learn to recognise the characteristics of each class.

1.2.1. Clock-based (cyclic, periodic) Tasks

A process plant operates in real time and thus we talk about the plant *time constants* (a time constant is a measure of the time taken by a plant to respond to a change in input or load and is used as a *characteristic* of the plant). The shorter the time constant of the process, the faster the required execution time. The computer which is used to control the plant must therefore be synchronised to real time and must be able to carry out all the required operations - measurement, control and actuation - within each sampling interval.

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. Synchronisation is usually obtained by adding a clock to the computer system normally referred to as *a real-time* clock - and using a signal from this clock to *interrupt* the operations of the computer at some predetermined fixed time interval.

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 systems the tasks may be subdivided into groups for controlling different parts of the plant and these may need to run at different sampling rates. The clock interrupt is frequently used to keep a clock and calendar so that the computer system is aware of both the time and the date. 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.

1.2.2. Event-based (aperiodic) Tasks

There are many systems where actions have to be performed not at particular times or time intervals but in response to some event. Typical examples are: turning off a pump or closing a valve when the level of a liquid in a tank reaches a predetermined value or switching a motor off in response to the closure of a microswitch indicating that some desired position has been reached. Event-based systems are also used extensively to indicate alarm conditions and initiate alarm actions, for example as an indication of too high a temperature or too great a pressure. The specification of event-based systems usually includes a requirement that the system must respond within a given maximum time to a particular event

Event-based systems normally employ interrupts to inform the computer system that .action is required. Some small, simple systems may use *polling;* that is, the computer periodically asks (polls) the various sensors to see if .action is required.

Events occur at non-deterministic intervals and event-based tasks are frequently referred to as *aperiodic* tasks. Such tasks may have deadlines expressed in terms 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 event.

1.2.3. Interactive Systems

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;

computerised 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 an 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 state 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. The test as to whether or not they are clock based as described above is to ask, `Can the system be tightly synchronised to an external process?' If the answer is `yes' they are clock based; if it is `no' then they are interactive.

1.3. Classification of Programs

Tasks of real-time systems are performed by real-time programs. They are programs for which the correctness of operation depends both on the logical results of the computation and the time at which the results are produced.

The importance of separating the various activities carried out by computer 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. As a result, the division of software into small, coherent *modules is* an important design technique and one of the guidelines for module division is to put activities with different types of time constraints into separate modules.

There are three main types of programs :

- sequential,
- multi-tasking,
- 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.

1.3.1. Sequential programs

In classical sequential programming *actions* are strictly ordered as a time sequence: the behaviour 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 that :
- a particular statement defines a stated action; and
- various program structures produce a stated sequence of events.

1.3.2. Multi-tasking programs

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 synchronisation 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). Note that the use of a synchronising procedure means the time taken for each individual action is no relevant to the verification procedure. The task can proceed at any speed: the correctness depends on the actions of the synchronising procedure

1.3.3. Real-Time programs

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 synchronisation rules. A real-time program can still be divided into : a number of tasks but communication between the tasks cannot necessarily :wait for a synchronisation 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.

Consideration of the types of reasoning necessary for the verification. of programs is important, not because we 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.