Introduction

Commercial usage of computer dates back to a little more than fifty years. This brief period can roughly be divided into mainframe, PC, and post-PC eras of computing. The mainframe era was marked by expensive computers that were quite unaffordable by individuals, and each computer served a large number of users. The PC era saw the emergence of desktops which could be easily be afforded and used by the individual users. The post-PC era is seeing emergence of small and portable computers, and computers embedded in everyday applications, making an individual interact with several computers everyday.

Real-time and embedded computing applications in the first two computing era were rather rare and restricted to a few specialized applications such as space and defense. In the post-PC era of computing, the use of computer systems based on real-time and embedded technologies has already touched every facet of our life and is still growing at a pace that was never seen before. While embedded processing and Internet-enabled devices have now captured everyone's imagination, they are just a small fraction of applications that have been made possible by real-time systems. If we casually look around us, we can discover many of them — often they are camouflaged inside simple looking devices. If we observe carefully, we can notice several gadgets and applications which have today become indispensable to our every day life, are in fact based on embedded real-time systems. For example, we have ubiquitous consumer products such as digital cameras, cell phones, microwave ovens, camcorders, video game sets; telecommunication domain products and applications such as set-top boxes, cable modems, voice over IP (VoIP), and video conferencing applications; office products such as fax machines, laser printers, and security systems. Besides, we encounter real-time systems in hospitals in the form of medical instrumentation equipments and imaging systems. There are also a large number of equipments and gadgets based on real-time systems which though we normally do not use directly, but never the less are still important to our daily life. A few examples of such systems are Internet routers, base stations in cellular systems, industrial plant automation systems, and industrial robots.

It can be easily inferred from the above discussion that in recent times real-time computers have become ubiquitous and have permeated large number of application areas. At present, the computers used in real-time applications vastly outnumber the computers that are being used in conventional applications. According to an estimate [3], 70% of all processors manufactured world-wide are deployed in real-time embedded applications. While it is already true that an overwhelming majority of all processors being manufactured are getting deployed in real-time applications, what is more remarkable is the unmistakable trend of steady rise in the fraction of all processors manufactured world-wide finding their way to real-time applications.

Some of the reasons attributable to the phenomenal growth in the use of real-time systems in the recent years are the manifold reductions in the size and the cost of the computers, coupled with the magical improvements to their performance. The availability of computers at rapidly falling prices, reduced weight, rapidly shrinking sizes, and their increasing processing power have together contributed to the present scenario. Applications which not too far back were considered prohibitively expensive to automate, can now be affordably automated. For instance, when microprocessors costed several tens of thousands of Rupees, they were considered to be too expensive to be put inside a washing machine; but when they cost only a few hundred rupees, their use makes commercial sense.

The rapid growth of applications deploying real-time technologies has been matched by the evolutionary growth of the underlying technologies supporting the development of real-time systems. In this book, we discuss some of the core technologies used in developing real-time systems. However, we restrict ourselves to software issues only and keep hardware discussions to bare minimum. The software issues that we address are quite expansive in the sense that besides the operating system and program development issues, we discuss the networking and database issues.

In this chapter, we restrict ourselves to some introductory and fundamental issues. In the next three chapters, we discuss some core theories underlying the development of practical real-time and embedded systems. In the subsequent chapter, we discuss some important features of commercial real-time operating systems. After that, we shift our attention to real-time communication technologies and databases.

1 What is Real Time?

Real time is a quantitative notion of time. Real-time is measured using a physical (real) clock. Whenever we quantify time using a physical clock, we deal with real time. An example use of this quantitative notion of time can be observed in a description of an automated chemical plant. Consider this: when the temperature of the chemical reaction chamber attains a certain predetermined temperature, say 250^o C, the system automatically switches off the heater within a predetermined time interval, say within 30 milli seconds. In this description of a part of the behavior of a chemical plant, the time value that was referred to denotes the readings of some physical clock present in the plant automation system.

In contrast to **real time**, **logical time** (also known as virtual time) deals with a qualitative notion of time and is expressed using event ordering relations such as before, after, sometimes, eventually, precedes, succeeds, etc. While dealing with logical time, time readings from a physical clock are not necessary for ordering the events. As an example, consider the following part of the behavior of a library automation software used to automate the book-keeping activities of a college library: "After a *query book* command is given by the user, details of all matching books are displayed by the software." In this example, the events "issue of query book command" and "display of results" are logically ordered in terms of which events follow the other. But, no quantitative expression of time was required. Clearly, this example behavior is devoid of any real-time considerations. We are now in a position to define what is a real-time system:

A system is called a **real-time system**, when we need quantitative expression of time (i.e. real-time) to describe the behavior of the system.

Remember that in this definition of a real-time system, it is implicit that all quantitative time measurements are carried out using a physical clock. A chemical plant, whose part behavior description is — `when temperature of the reaction chamber attains certain predetermined temperature value, say` 250^o `C, the system automatically switches off the heater within say 30 milli seconds` — is clearly a real-time system. Our examples so far were restricted to the description of partial behavior of systems. The complete behavior of a system can be described by listing its response to various external stimuli. It may be noted that all the clauses in the description of the behavior of a real-time system need not involve quantitative measures of time. That is, large parts of a description of the behavior of a system may not have any quantitative expressions of time at all, and still qualify as a real-time system. Any system whose behavior can completely be described without using any quantitative expression of time is of course not a real-time system.

2 Applications of Real-Time Systems

Real-time systems have of late, found applications in wide ranging areas. In the following, we list some of the prominent areas of application of real-time systems and in each identified case, we discuss a few example applications in some detail. As we can imagine, the list would become very vast if we try to exhaustively list all areas of applications of real-time systems. We have therefore restricted our list to only a handful of areas, and out of these we have explained only a few selected applications to conserve space. We have pointed out the quantitative notions of time used in the discussed applications. The examples we present are important to our subsequent discussions and would be referred to in the later Chapters whenever required.

Industrial Applications: Industrial applications constitute a major usage area of real-time systems. A few examples of industrial applications of real-time systems are: process control systems, industrial automation systems, SCADA applications, test and measurement equipments, and robotic equipments.

Example 1: Chemical Plant Control
Chemical plant control systems are essentially a type of process control application. In an automated chemical plant, a real-time computer periodically monitors plant conditions. The plant conditions are determined based on current readings of pressure, temperature, and chemical concentration of the reaction chamber. These parameters are sampled periodically. Based on the values sampled at any time, the automation system decides on the corrective actions necessary at that instant to maintain the chemical reaction at a certain rate. Each time the plant conditions are sampled, the automation system should decide on the exact instantaneous corrective actions required such as changing the pressure, temperature, or chemical concentration and carry out these actions within certain predefined time bounds. Typically, the time bounds in such a chemical plant control application range from a few micro seconds to several milli seconds.

Example 2: Automated Car Assembly Plant
An automated car assembly plant is an example of a plant automation system. In an automated car assembly plant, the work product (partially assembled car) moves on a conveyor belt (see Fig. 1). Alongside the conveyor belt, several workstations are placed. Each workstation performs some specific work on the work product such as fitting engine, fitting door, fitting wheel, and spray painting the car, etc. as it moves on the conveyor belt. An empty chassis is introduced near the first workstation on the conveyor belt. A fully assembled car comes out after the work product goes past all the workstations. At each workstation, a sensor senses the arrival of the next partially assembled product. As soon as the partially assembled product is sensed, the workstation begins to perform its work on the work product. The time constraint imposed on the workstation computer is that the workstation must complete its work before the work product moves away to the next workstation. The time bounds involved here are typically of the order of a few hundreds of milli seconds.

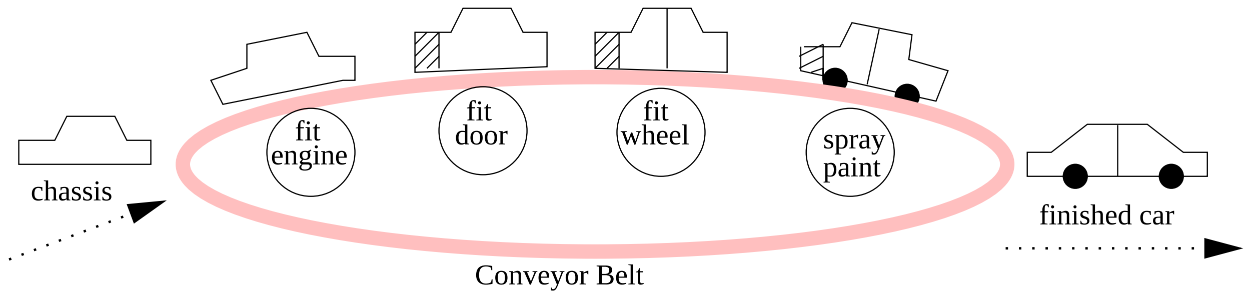

Figure 1: Schematic Representation of An Automated Car Assembly Plant

Example 3: Supervisory Control And Data Acquisition(SCADA)
SCADA are a category of distributed control systems being used in many industries. A SCADA system helps monitor and control a large number of distributed events of interest. In SCADA systems, sensors are scattered at various geographic locations to collect raw data (called events of interest). These data are then processed and stored in a real-time database. The database models (or reflects) the current state of the environment. The database is updated frequently to make it a realistic model of the uptodate state of the environment. An example of a SCADA application is an Energy Management System(EMS). An EMS helps to carry out load balancing in an electrical energy distribution network. The EMS senses the energy consumption at the distribution points and computes the load across different phases of power supply. It also helps dynamically balance

the load. Another example of a SCADA system is a system that monitors and controls traffic in a computer network. Depending on the sensed load in different segments of the network, the SCADA system makes the router change its traffic routing policy dynamically. The time constraint in such a SCADA application is that the sensors must sense the system state at regular intervals (say every few milli seconds) and the same must be processed before the next state is sensed.

Medical: A few examples of medical applications of real-time systems are: robots, MRI scanners, radiation therapy equipments, bedside monitors, and computerized axial tomography (CAT).

Example 4: Robot Used in Recovery of Displaced Radioactive Material
Robots have become very popular nowadays and are being used in a wide variety of medical applications. An application that we discuss here is a robot used in retrieving displaced radioactive materials. Radioactive materials such as Cobalt and Radium are used for treatment of cancer. At times during treatment, the radioactive Cobalt (or Radium) gets dislocated and falls down. Since human beings can not come near a radioactive material, a robot is used to restore the radioactive material to its proper position. The robot walks into the room containing the radioactive material, picks it up, and restores it to its proper position. The robot has to sense its environment frequently and based on this information, plan its path. The real-time constraint on the path planning task of the robot is that unless it plans the path fast enough after an obstacle is detected, it may collide with it. The time constraints involved here are of the order of a few milli seconds.

Peripheral equipments: A few examples of peripheral equipments that contain embedded real-time systems are: laser printers, digital copiers, fax machines, digital cameras, and scanners.

Example 5: Laser Printer
Most laser printers have powerful microprocessors embedded in them to control different activities associated with printing. The important activities that a microprocessor embedded in a laser printer performs include the following: getting data from the communication port(s), typesetting fonts, sensing paper jams, noticing when the printer runs out of paper, sensing when the user presses a button on the control panel, and displaying various messages to the user. The most complex activity that the microprocessor performs is driving the laser engine. The basic command that a laser engine supports is to put a black dot on the paper. However, the laser engine has no idea about the exact shapes of different fonts, font sizes, italic, underlining, boldface, etc. that it may be asked to print. The embedded microprocessor receives print commands on its input port and determines how the dots can be composed to achieve the desired document and manages printing the exact shapes through a series of dot commands issued to the laser engine. The time constraints involved here are of the order of a few milli seconds.

Automotive and Transportation: A few examples of automotive and transportation applications of real-time systems are: automotive engine control systems, road traffic signal control, air-traffic control, high-speed train control, car navigation systems, and MPFI engine control systems.

Example 6: Multi-Point Fuel Injection (MPFI) System
An MPFI system is an automotive engine control system. A conceptual diagram of a car embedding an MPFI system is shown in Fig. 2. An MPFI is a real-time system that controls the rate of fuel injection and allows the engine to operate at its optimal efficiency. In older models of cars, a mechanical device called the carburettor was used to control the fuel injection rate to the engine. It was the responsibility of the carburettor to vary the fuel injection rate depending on the current speed of the vehicle and the desired acceleration. Careful experiments have suggested that for optimal energy output, the required fuel injection rate is highly nonlinear with respect to the vehicle speed and acceleration. Also, experimental results show that the precise fuel injection through multiple points is more effective than single point injection. In MPFI engines, the precise fuel injection rate at each injection point is determined by a computer. An MPFI system injects fuel into individual cylinders resulting in better 'power balance' among the cylinders as well as higher output from each one along with faster throttle response. The processor primarily controls the ignition timing and the quantity of fuel to be injected. The latter is achieved by controlling the duration for which the injector valve is open — popularly known as

pulse width. The actions of the processor are determined by the data gleaned from sensors located all over the engine. These sensors constantly monitor the ambient temperature, the engine coolant temperature, exhaust temperature, emission gas contents, engine rpm (speed), vehicle road speed, crankshaft position, camshaft position, etc. An MPFI engine with even an 8-bit computer does a much better job of determining an accurate fuel injection rate for given values of speed and acceleration compared to a carburettor-based system. An MPFI system not only makes a vehicle more fuel efficient, it also minimizes pollution by reducing partial combustion.

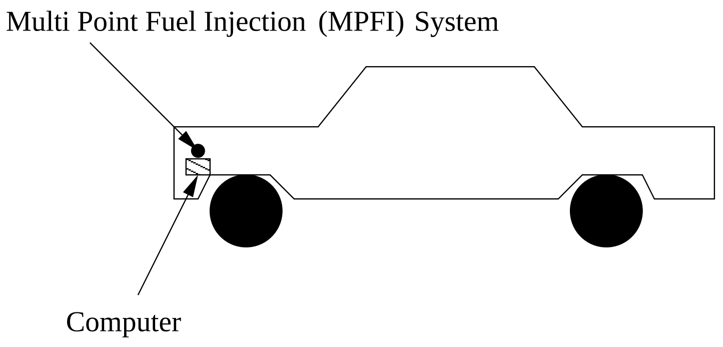

Figure 2: A Real-Time System Embedded in An MPFI Car

Telecommunication Applications: A few example uses of real-time systems in telecommunication applications are: cellular systems, video conferencing, and cable modems.

Example 7: A Cellular System
Cellular systems have become a very popular means of mobile communication. A cellular system usually maps a city into cells. In each cell, a base station monitors the mobile handsets present in the cell. Besides, the base station performs several tasks such as locating a user, sending and receiving control messages to a handset, keeping track of call details for billing purposes, and hand-off of calls as the mobile moves. Call hand-off is required when a mobile moves away from a base station. As a mobile moves away, its received signal strength (RSS) falls at the base station. The base station monitors this and as soon as the RSS falls below a certain threshold value, it hands-off the details of the on-going call of the mobile to the base station of the cell to which the mobile has moved. The hand-off must be completed within a sufficiently small predefined time interval. so that the user does not feel any temporary disruption of service during the hand-off. Typically call hand-off is required to be achieved within a few milliseconds.

Aerospace: A few important use of real-time systems in aerospace applications are: avionics, flight simulation, airline cabin management systems, satellite tracking systems, and computer on-board an aircraft.

Example 8: Computer On-board an Aircraft
In many modern aircrafts, the pilot can select an "auto pilot" option. As soon as the pilot switches to the "auto pilot" mode, an on-board computer takes over all controls of the aircraft including navigation, take-off, and landing of the aircraft. In the "auto pilot" mode, the computer periodically samples velocity and acceleration of the aircraft. From the sampled data, the on-board computer computes X, Y, and Z co-ordinates of the current aircraft position and compares them with the pre-specified track data. Before the next sample values are obtained, it computes the deviation from the specified track values and takes any corrective actions that may be necessary. In this case, the sampling of the various parameters, and their processing need to be completed within a few micro seconds.

Internet and Multimedia Applications: Important use of real-time systems in multimedia and Internet applications include: video conferencing and multimedia multicast, Internet routers and switches.

Example 9: Video Conferencing
In a video conferencing application, video and audio signals are generated by cameras and microphones respectively. The data are sampled at a certain prespecified frame rate. These are then compressed and sent

as packets to the receiver over a network. At the receiver-end, packets are ordered, decompressed, and then played. The time constraint at the receiver-end is that the receiver must process and play the received frames at a predetermined constant rate. Thus if thirty frames are to be shown every minute, once a frame play-out is complete, the next frame must be played within two seconds.

Consumer Electronics: Consumer electronics area abounds numerous applications of real-time systems. A few sample applications of real-time systems in consumer electronics are: set-top boxes, audio equipment, Internet telephony, microwave ovens, intelligent washing machines, home security systems, air conditioning and refrigeration, toys, and cell phones.

Example 10: Cell Phones
Cell phones are possibly the fastest growing segment of consumer electronics. A cell phone at any point of time carries out a number of tasks simultaneously. These include: converting input voice to digital signals by deploying digital signal processing (DSP) techniques, converting electrical signals generated by the microphone to output voice signals, and sampling incoming base station signals in the control channel. A cell phone responds to the communications received from the base station within certain specified time bounds. For example, a base station might command a cell phone to switch the on-going communication to a specific frequency. The cell phone must comply to such commands from the base station within a few milli seconds.

Defense Applications: Typical defense applications of real-time systems include: missile guidance systems, anti-missile systems, satellite-based surveillance systems.

Example 11: Missile Guidance System
A guided missile is one that is capable of sensing the target and homes onto it. Homing becomes easy when the target emits either electrical or thermal radiation. In a missile guidance system, missile guidance is achieved by a computer mounted on the missile. The mounted computer computes the deviation from the required trajectory and effects track changes of the missile to guide it onto the target. The time constraint on the computer-based guidance system is that the sensing and the track correction tasks must be activated frequently enough to keep the missile from diverging from the target. The target sensing and track correction tasks are typically required to be completed within a few hundreds of microseconds or even lesser time depending on the speed of the missile and the type of the target.

Miscellaneous Applications: Besides the areas of applications already discussed, real-time systems have found numerous other applications in our every day life. An example of such an application is a railway reservation system.

Example 12: Railway Reservation System
In a railway reservation system, a central repository maintains the up-to-date data on booking status of various trains. Ticket booking counters are distributed across different geographic locations. Customers queue up at different booking counters and submit their reservation requests. After a reservation request is made at a counter, it normally takes only a few seconds for the system to confirm the reservation and print the ticket. A real-time constraint in this application is that once a request is made to the computer, it must print the ticket or display the seat unavailability message before the average human response time (about 20 seconds) expires, so that the customers do not notice any delay and get a feeling of having obtained instant results. However, as we discuss a little later (in Sec. 1.6), this application is an example of a category of applications that is in some aspects different from the other discussed applications. For example, even if the results are produced just after 20 Secs, nothing untoward is going to happen — this may not be the case with the other discussed applications.

3 A Basic Model of a Real-Time System

We have already pointed out that this book confines itself to the software issues in real-time systems. However, in order to be able to see the software issues in a proper perspective, we need to have a basic conceptual understanding of the underlying hardware. We therefore in this section try to develop a broad understanding of high level issues of the underlying hardware in a real-time system. For a more detailed study of the underlying hardware issues, we refer the reader to [2]. Fig. 3 shows a simple model of a real-time system in terms of its important functional blocks. Unless otherwise mentioned, all our subsequent discussions would implicitly assume such a model. Observe that in Fig. 3, the sensors are interfaced with the input conditioning block, which in turn is connected to the input interface. The output interface, output conditioning, and the actuator are interfaced in a complementary manner. In the following, we briefly describe the roles of the different functional blocks of a real-time system:

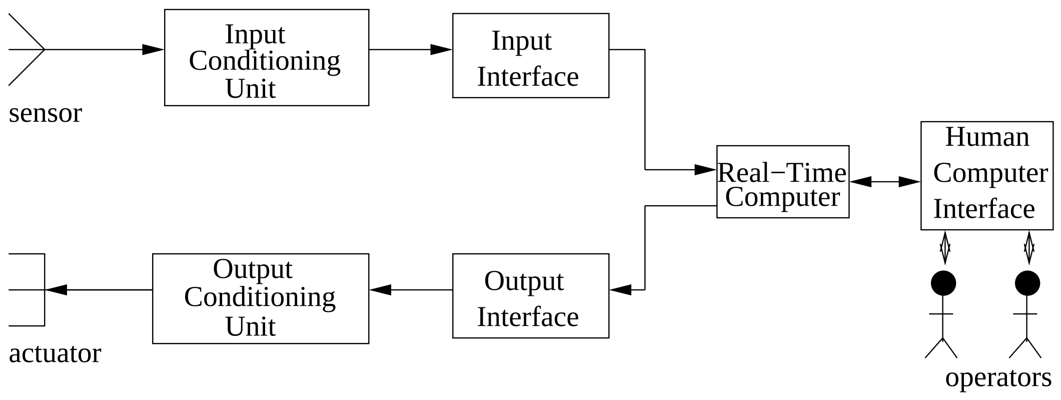

Figure 3: A Model of a Real-Time System

Sensor: A sensor converts some physical characteristic of its environment into electrical signals. An example of a sensor is a photo-voltaic cell which converts light energy into electrical energy. A wide variety of temperature and pressure sensors are also used. A temperature sensor typically operates based on the principle of a **thermocouple**. Temperature sensors based on many other physical principles also exist. For example, one type of temperature sensor employs the principle of variation of electrical resistance with temperature (called a *varistor*). A pressure sensor typically operates based on the *piezoelectricity* principle. Pressure sensors based on other physical principles also exist.

Actuator: An actuator is any device that takes its inputs from the output interface of a computer and converts these electrical signals into some physical actions on its environment. The physical actions may be in the form of motion, change of thermal, electrical, pneumatic, or physical characteristics of some objects. A popular actuator is a motor. Heaters are also very commonly used. Besides, several hydraulic and pneumatic actuators are also popular.

Signal Conditioning Units: The electrical signals produced by a computer can rarely be used to directly drive an actuator. The computer signals usually need conditioning before they can be used by the actuator. This is termed *output conditioning*. Similarly, input conditioning is required to be carried out on sensor signals before they can be accepted by the computer. For example, analog signals generated by a photo-voltaic cell are normally in the milli volts range and need to be conditioned before they can be processed by a computer. The following are some important types of conditioning carried out on raw signals generated by sensors and digital signals generated by computers:

1. **Voltage Amplification:** Voltage amplification is normally required to be carried out to match the full scale sensor voltage output with the full scale voltage input to the interface of a computer. For example, a sensor might produce voltage in the milli volts range, whereas the input interface of a computer may require the input signal level to be of the order of a volt.

2. **Voltage Level Shifting:** Voltage level shifting is often required to align the voltage level generated by a sensor with that acceptable to the computer. For example, a sensor may produce voltage in the range -0.5 to +0.5 volt, whereas the input interface of the computer may accept voltage only in the range of 0 to 1 volt. In this case, the sensor voltage must undergo level shifting before it can be used by the computer.

3. **Frequency Range Shifting and Filtering:** Frequency range shifting is often used to reduce the noise components in a signal. Many types of noise occur in narrow bands and the signal must be shifted from the noise bands so that noise can be filtered out.

4. **Signal Mode Conversion:** A type of signal mode conversion that is frequently carried out during signal conditioning involves changing direct current into alternating current and vice-versa. Another type signal mode conversion that is frequently used is conversion of analog signals to a constant amplitude pulse train such that the pulse rate or pulse width is proportional to the voltage level. Conversion of analog signals to a pulse train is often necessary for input to systems such as **transformer coupled circuits** that do not pass direct current.

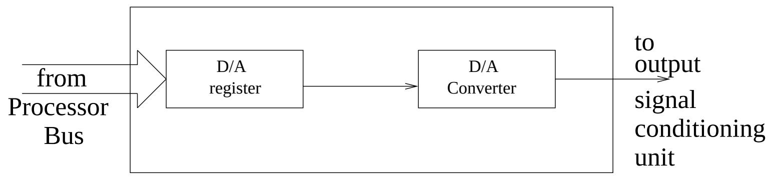

Figure 4: An Output Interface

Interface Unit: Normally commands from the CPU are delivered to the actuator through an output interface. An output interface converts the stored voltage into analog form and then outputs this to the actuator circuitry. This of course would require the value generated to be written on a register (see Fig. 4). In an output interface, in order to produce an analog output, the CPU selects a data register of the output interface and writes the necessary data to it. The two main functional blocks of an output interface are shown in Fig. 4. The interface takes care of the buffering and the handshake control aspects. Analog to digital conversion is frequently deployed in an input interface. Similarly, digital to analog conversion is frequently used in an output interface.

In the following, we discuss the important steps of analog to digital signal conversion (ADC).

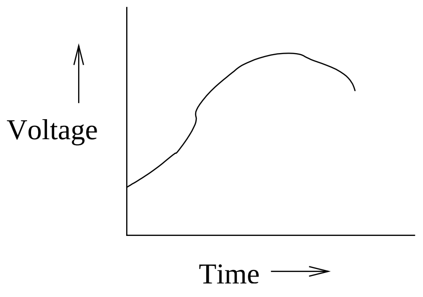

Figure 5: Continuous Analog Voltage

Analog to Digital Conversion:
Digital computers can not process analog signals. Therefore, analog signals need to be converted to digital form. Analog signals can be converted to digital form using a circuitry whose block diagram is shown in Fig. 7. Using the block diagram shown in Fig. 7, analog signals are normally converted to digital form through the following two main steps:

- Sample the analog signal (shown in Fig. 5) at regular intervals. This sampling can be done by a capacitor circuitry that stores the voltage levels. The stored voltage level can be discretized. After sampling the analog signal (shown in Fig. 5), a step waveform as shown in Fig. 6 is obtained.

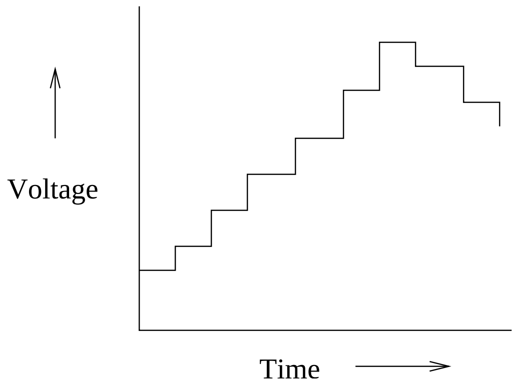

Figure 6: Analog Voltage Converted to Discrete Form

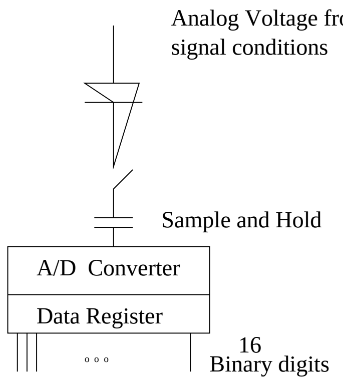

Figure 7: Conversion of an Analog Signal to a 16 bit Binary Number

- Convert the stored value to a binary number by using an analog to digital converter (ADC) as shown in Fig. 7, and store the digital value in a register.

Digital to analog conversion can be carried out through a complementary set of operations. We leave it as an exercise to the reader to figure out the details of the circuitry that can perform the digital to analog conversion (DAC).

4 Characteristics of Real-Time Systems

We now discuss a few key characteristics of real-time systems. These characteristics distinguish real-time systems from non-real-time systems. However, the reader may note that all the discussed characteristics may not be applicable to every real-time system. Real-time systems cover such an enormous range of applications and products that a generalization of the characteristics into a set that is applicable to each and every system is difficult. Different categories of real-time systems may exhibit the characteristics that we identify to different extents or may not even exhibit some of the characteristics at all.

1. **Time constraints.** Every real-time task is associated with some time constraints. One form of time constraints that is very common is deadlines associated with tasks. A task deadline specifies the time before which the task must complete and produce the results. Other types of timing constraints are delay and duration (see Section 1.7). It is the responsibility of the real-time operating system (RTOS) to ensure that all tasks meet

their respective time constraints. We shall examine in later chapters how an RTOS can ensure that tasks meet their respective timing constraints through appropriate task scheduling strategies.

2. **New Correctness Criterion.** The notion of correctness in real-time systems is different from that used in the context of traditional systems. In real-time systems, correctness implies not only logical correctness of the results, but the time at which the results are produced is important. A logically correct result produced after the deadline would be considered as an incorrect result.

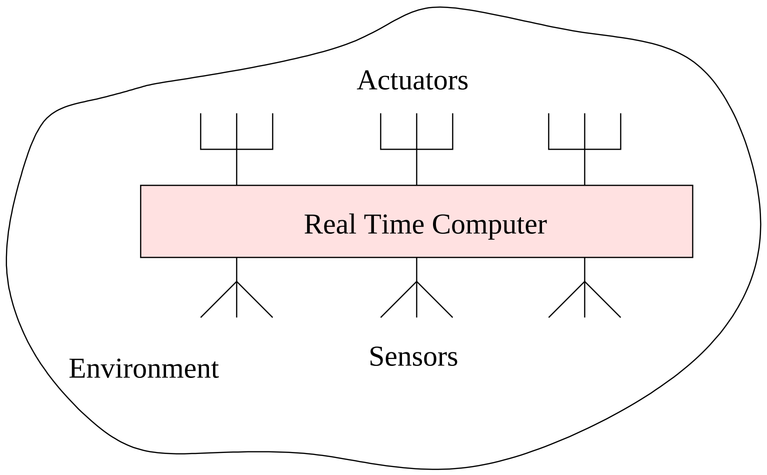

Figure 8: A Schematic Representation of An Embedded Real-Time System

3. **Embedded.** A vast majority of real-time systems are embedded in nature [3]. An embedded computer system is physically "embedded" in its environment and often controls it. Fig. 8 shows a schematic representation of an embedded system. As shown in Fig. 8, the sensors of the real-time computer collect data from the environment, pass them on to the real-time computer for processing. The computer, in turn passes information (processed data) to the actuators to carry out the necessary work on the environment, which results in controlling some characteristics of the environment. Several examples of embedded systems were discussed in Sec. 1.2. An example of an embedded system that we would often refer is the Multi-Point Fuel Injection (MPFI) system discussed in Example 6 of Sec. 1.2.

4. **Safety-Criticality.** For traditional non-real-time systems safety and reliability are independent issues. However, in many real-time systems these two issues are intricately bound together making them *safety-critical*. Note that a **safe** system is one that does not cause any damage even when it fails. A **reliable** system on the other hand, is one that can operate for long durations of time without exhibiting any failures. A safety-critical system is required to be highly reliable since any failure of the system can cause extensive damages. We elaborate this issue in Section 1.5.

5. **Concurrency.** A real-time system usually needs to respond to several independent events within very short and strict time bounds. For instance, consider a chemical plant automation system (see Example 1 of Sec. 1.2), which monitors the progress of a chemical reaction and controls the rate of reaction by changing the different parameters of reaction such as pressure, temperature, and chemical concentration. These parameters are sensed using sensors fixed in the chemical reaction chamber. These sensors may generate data asynchronously at different rates. Therefore, the real-time system must process data from all the sensors concurrently, otherwise signals may be lost and the system may malfunction. These systems can be considered to be non-deterministic, since the behavior of the system depends on the exact timing of its inputs. A non-deterministic computation is one in which two runs using the same set of input data can produce two distinct sets of output data in the two runs.

6. **Distributed and Feedback Structure.** In many real-time systems, the different components of the system are naturally distributed across widely spread geographic locations. In such systems, the different events of interest arise at the geographically separate locations. Therefore, these events may often have to be handled locally and responses produced to them to prevent overloading of the underlying communication network. Therefore, the sensors and the actuators may be located at the places where events are generated. An example of such a system is a petroleum refinery plant distributed over a large geographic area. At each data source, it makes good design sense to locally process the data before being passed on to a central processor.

   Many distributed as well as centralized real-time systems have a feedback structure as shown in Fig. 9. In these systems, the sensors usually sense the environment periodically. The sensed data about the environment is processed to determine the corrective actions necessary. The results of the processing are used to carry out the necessary corrective actions on the environment through the actuators, which in turn again cause a change to the required characteristics of the controlled environment, and so on.

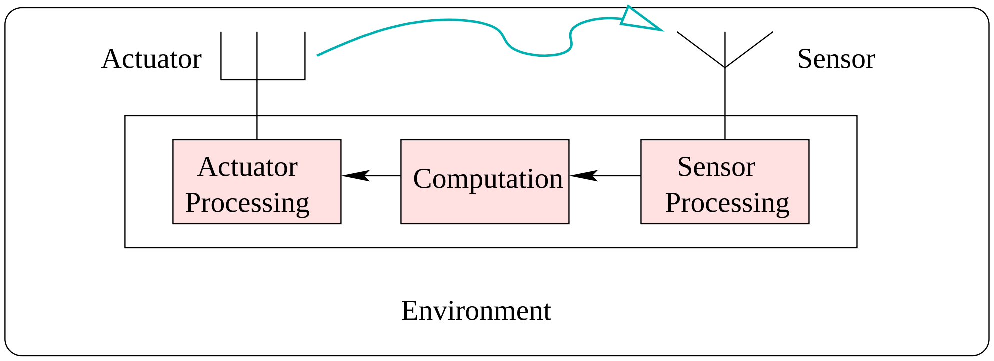

Figure 9: Feedback Structure of Real-Time Systems

7. **Task Criticality.** Task criticality is a measure of the cost of failure of a task. Task criticality is determined by examining how critical are the results produced by the task to the proper functioning of the system. A real-time system may have tasks of very different criticalities. It is therefore natural to expect that the criticalities of the different tasks must be taken into consideration while designing for fault-tolerance. The higher the criticality of a task, the more reliable it should be made. Further, in the event of a failure of a highly critical task, immediate failure detection and recovery are important. However, it should be realized that task priority is a different concept and task criticality does not solely determine the task priority or the order in which various tasks are to be executed (these issues shall be elaborated in the later chapters).

8. **Custom Hardware.** A real-time system is often implemented on custom hardware that is specifically designed and developed for the purpose. For example, a cell phone does not use traditional microprocessors. Cell phones use processors which are tiny, supporting only those processing capabilities that are really necessary for cell phone operation and specifically designed to be power-efficient to conserve battery life. The capabilities of the processor used in a cell phone are substantially different from that of a general purpose processor. Another example is the embedded processor in an MPFI car. In this case, the processor used need not be a powerful general purpose processor such as a Pentium or an Athlon processor. Some of the most powerful computers used in MPFI engines are 16- or 32-bit processors running at approximately 40 MHz. However, unlike the conventional PCs, a processor used in these car engines do not deal with processing frills such as screen-savers or a dozen of different applications running at the same time. All that the processor in an MPFI system needs to do is to compute the required fuel injection rate that is most efficient for a given speed and acceleration.

9. **Reactive.** Real-time systems are often *reactive*. A reactive system is one in which an on-going interaction between the computer and the environment is maintained. Ordinary systems compute functions on the input

data to generate the output data (See Fig. 10 (a)). In other words, traditional systems compute the output data as some function ϕ of the input data. That is, output data can mathematically be expressed as: $output\ data = \phi(input\ data)$. For example, if some data I_1 is given as the input, the system computes O_1 as the result$O_1 = \phi(I_1)$. To elaborate this concept, consider an example involving a library automation software. In a library automation software, when the query book function is invoked and "Real-Time Systems" is entered as the input book name, then the software displays "Author name: R. Mall, Rack Number: 001, Number of Copies: 1".

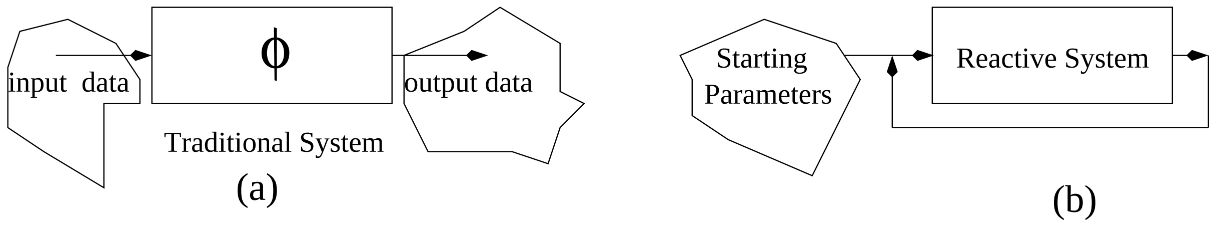

Figure 10: Traditional versus Reactive Systems

In contrast to the traditional computation of the output as a simple function of the input data, real-time systems do not produce any output data but enter into an on-going interaction with their environment. In each interaction step, the results computed are used to carry out some actions on the environment. The reaction of the environment is sampled and is fed back to the system. Therefore the computations in a real-time system can be considered to be non-terminating. This reactive nature of real-time systems is schematically shown in the Fig. 10 (b).

10. **Stability:** Under overload conditions, real-time systems need to continue to meet the deadlines of the most critical tasks, though the deadlines of non-critical tasks may not be met. This is in contrast to the requirement of *fairness* for traditional systems even under overload conditions.

11. **Exception Handling:** Many real-time systems work round-the-clock and often operate without human operators. For example, consider a small automated chemical plant that is set up to work non-stop. When there are no human operators, taking corrective actions on a failure becomes difficult. Even if no corrective actions can be immediate taken, it is desirable that a failure does not result in catastrophic situations. A failure should be detected and the system should continue to operate in a gracefully degraded mode rather than shutting off abruptly.

5 Safety and Reliability

In traditional systems, safety and reliability are normally considered to be independent issues. It is therefore possible to identify a traditional system that is safe and unreliable and systems that are reliable but unsafe. Consider the following two examples. A word processing software may not be very reliable but is safe. A failure of the software does not usually cause any significant damage or financial loss. It is therefore an example of an unreliable but safe system. On the other hand, a hand gun can be unsafe but is reliable. A hand gun rarely fails. A hand gun is an unsafe system because if it fails for some reason, it can misfire or even explode and cause significant damage. It is an example of an unsafe but reliable system. These two examples show that for traditional systems, safety and reliability are independent concerns — it is therefore possible to increase the safety of a system without affecting its reliability and vice versa.

In real-time systems on the other hand, safety and reliability are coupled together. Before analyzing why safety and reliability are no longer independent issues in real-time systems, we need to first understand what exactly is meant by a **fail-safe state**.

A fail-safe state of a system is one which if entered when the system fails, no damage would result.

To give an example, the fail-safe state of a word processing program is one where the document being processed has been saved onto the disk. All traditional non real-time systems do have one or more fail-safe states which help separate the issues of safety and reliability — even if a system is known to be unreliable, it can always be made to fail in a fail-safe state, and consequently it would still be considered to be a safe system.

If no damage can result if a system enters a fail-safe state just before it fails, then through careful transit to a fail-safe state upon a failure, it is possible to turn an extremely unreliable and unsafe system into a safe system. In many traditional systems this technique is in fact frequently adopted to turn an unreliable system into a safe system. For example, consider a traffic light controller that controls the flow of traffic at a road intersection. Suppose the traffic light controller fails frequently and is known to be highly unreliable. Though unreliable, it can still be considered safe if whenever a traffic light controller fails, it enters a fail-safe state where all the traffic lights are orange and blinking. This is a **fail-safe state,** since the motorists on seeing blinking orange traffic light become aware that the traffic light controller is not working and proceed with caution. Of course, a fail-safe state may not be to make all lights green, in which case severe accidents could occur. Similarly, all lights turned red is also not a fail-safe state — it may not cause accidents, but would bring all traffic to a stand still leading to traffic jams. However, in many real-time systems there are no fail-safe states. Therefore, any failure of the system can cause severe damages. Such systems are said to be **safety-critical systems**.

A safety-critical system is one whose failure can cause severe damages.

An example of a safety-critical system is a navigation system on-board an aircraft. An on-board navigation system has no fail-safe states. When the computer on-board an aircraft fails, a fail-safe state may not be one where the engine is switched-off! In a safety-critical system, the absence of fail-safe states implies that safety can only be ensured through increased reliability. Thus, for safety-critical systems the issues of safety and reliability become interrelated — safety can only be ensured through increased reliability. It should now be clear why safety-critical systems need to be highly reliable.

Just to give an example of the level of reliability required of safety-critical systems, consider the following. For any fly-by-wire aircraft, most of its vital parts are controlled by a computer. Any failure of the controlling computer is clearly not acceptable. The standard reliability requirement for such aircrafts is at most 1 failure per 10^9 flying hours (that is, a million years of continuous flying!). We examine how a highly reliable system can be developed in the next section.

5.1 How to Achieve High Reliability?

If you are asked by your organization to develop a software which should be highly reliable, how would you proceed to achieve it? Highly reliable software can be developed by adopting all of the following three important techniques:

- **Error Avoidance.** For achieving high reliability, every possibility of occurrence of errors should be minimized during product development as much as possible. This can be achieved by adopting a variety of means: using well-founded software engineering practices, using sound design methodologies, adopting suitable CASE tools, and so on.
- **Error Detection and Removal.** In spite of using the best available error avoidance techniques, many errors still manage to creep into the code. These errors need to be detected and removed. This can be achieved to a large extent by conducting thorough reviews and testing. Once errors are detected, they can be easily fixed.

- **Fault-Tolerance.** No matter how meticulously error avoidance and error detection techniques are used, it is virtually impossible to make a practical software system entirely error-free. Few errors still persist even after carrying out thorough reviews and testing. Errors cause failures. That is, failures are manifestation of the errors latent in the system. Therefore to achieve high reliability, even in situations where errors are present, the system should be able to tolerate the faults and compute the correct results. This is called fault-tolerance. Fault-tolerance can be achieved by carefully incorporating redundancy.

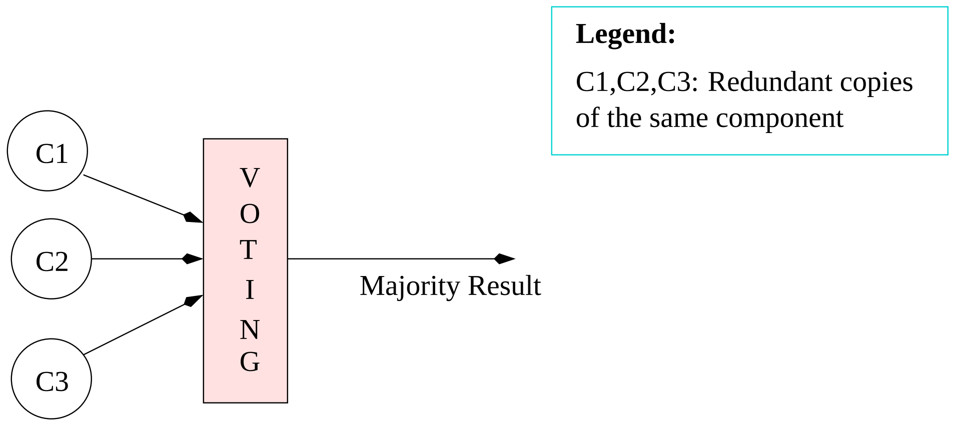

Figure 11: Schematic Representation of TMR

It is relatively simple to design a hardware equipment to be fault-tolerant. The following are two methods that are popularly used to achieve hardware fault-tolerance:

- **Built In Self Test (BIST)**: In BIST, the system periodically performs self tests of its components. Upon detection of a failure, the system automatically reconfigures itself by switching out the faulty component and switching in one of the redundant good components.
- **Triple Modular Redundancy (TMR)**: In TMR, as the name suggests, three redundant copies of all critical components are made to run concurrently (see Fig. 11). Observe that in Fig. 11, C1, C2, and C3 are the redundant copies of the same critical component. The system performs voting of the results produced by the redundant components to select the majority result. TMR can help tolerate occurrence of only a single failure at any time. (Can you answer why a TMR scheme can effectively tolerate a single component failure only?) An assumption that is implicit in the TMR technique is that at any time only one of the three redundant components can produce erroneous results. The majority result after voting would be erroneous if two or more components can fail simultaneously (more precisely, before a repair can be carried out). In situations where two or more components are likely to fail (or produce erroneous results), then greater amounts of redundancies would be required to be incorporated. A little thinking can show that at least 2n + 1 redundant components are required to tolerate simultaneous failures of n component.

As compared to hardware, software fault-tolerance is much harder to achieve. To investigate the reason behind this, let us first discuss the techniques currently being used to achieve software fault-tolerance. We do this in the following subsection.

5.2 Software Fault-Tolerance Techniques

Two methods are now popularly being used to achieve software fault-tolerance: N-version programming and recovery block techniques. These two techniques are simple adaptations of the basic techniques used to provide hardware fault-tolerance. We discuss these two techniques in the following.

N-Version Programming. This technique is an adaptation of the TMR technique for hardware fault-tolerance. In the N-version programming technique, independent teams develop N different versions (value of N depends on the degree of fault-tolerance required) of a software component (module). The redundant modules are run concurrently (possibly on redundant hardware). The results produced by the different versions of the module are subjected to voting at run time and the result on which majority of the components agree is accepted. The central idea behind this scheme is that independent teams would commit different types of mistakes, which would be eliminated when the results produced by them are subjected to voting. However, this scheme is not very successful in achieving fault-tolerance, and the problem can be attributed to *statistical correlation of failures.* Statistical correlation of failures means that even though individual teams worked in isolation to develop the different versions of a software component, still the different versions fail for identical reasons. In other words, the different versions of a component show similar failure patterns. This does mean that the different modules developed by independent programmers, after all, contain identical errors. The reason for this is not far to seek, programmers commit errors in those parts of a problem which they perceive to be difficult — and what is difficult to one team is usually difficult to all teams. So, identical errors remain in the most complex and least understood parts of a software component.

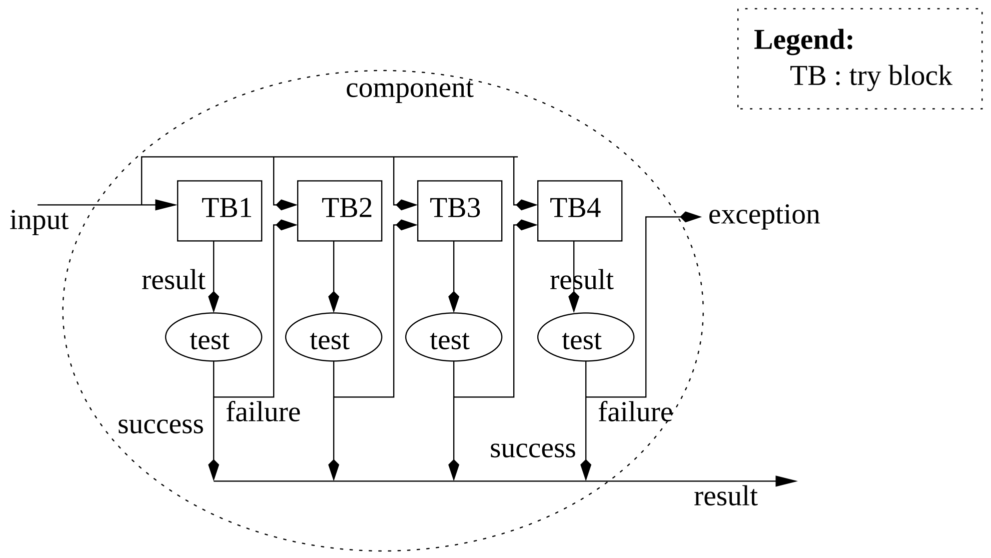

Figure 12: A Software Fault-Tolerance Scheme Using Recovery Blocks

Recovery Blocks. In the recovery block scheme, the redundant components are called *try blocks.* Each try block computes the same end result as the others but is intentionally written using a different algorithm compared to the other try blocks. In N-version programming, the different versions of a component are written by different teams of programmers, whereas in recovery block different algorithms are used in different try blocks. Also, in contrast to the N-version programming approach where the redundant copies are run concurrently, in the recovery block approach they are (as shown in Fig. 12) run one after another. The results produced by a try block are subjected to an acceptance test (see Fig. 12). If the test fails, then the next try block is tried. This is repeated in a sequence until the result produced by a try block successfully passes the acceptance test. Note that in Fig. 12 we have shown acceptance tests separately for different try blocks to help understand that the tests are applied to the try blocks one after the other, though it may be the case that the same test is applied to each try block.

As was the case with N-version programming, the recovery blocks approach also does not achieve much success in providing effective fault-tolerance. The reason behind this is again statistical correlation of failures. Different try blocks fail for identical reasons as was explained in case of N-version programming approach. Besides, this approach suffers from a further limitation that it can only be used if the task deadlines are much larger than the task com-

putation times (i.e. tasks have large laxity), since the different try blocks are put to execution one after the other when failures occur. The recovery block approach poses special difficulty when used with real-time tasks with very short slack time (i.e. short deadline and considerable execution time), as the try blocks are tried out one after the other deadlines may be missed. Therefore, in such cases the later try-blocks usually contain only skeletal code.
Of course, it is possible that the later try blocks contain only skeletal code, produce only approximate results and therefore take much less time for computation than the first try block.

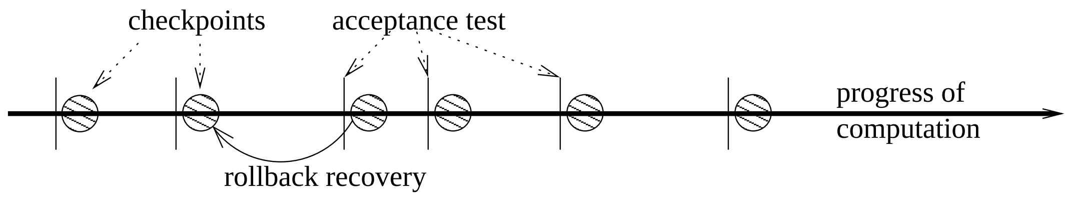

Figure 13: Checkpointing and Rollback Recovery

Checkpointing and Rollback Recovery:
Checkpointing and roll-back recovery is another popular technique to achieve fault-tolerance. In this technique as the computation proceeds, the system state is tested each time after some meaningful progress in computation is made. Immediately after a state-check test succeeds, the state of the system is backed up on a stable storage (see Fig. 13). In case the next test does not succeed, the system can be made to roll-back to the last checkpointed state. After a rollback, from a checkpointed state a fresh computation can be initiated. This technique is especially useful, if there is a chance that the system state may be corrupted as the computation proceeds, such as data corruption or processor failure.

6 Types of Real-Time Tasks

We have already seen that a real-time task is one for which quantitative expressions of time are needed to describe its behavior. This quantitative expression of time usually appears in the form of a constraint on the time at which the task produces results. The most frequently occurring timing constraint is a deadline constraint which is used to express that a task is required to compute its results within some deadline. We therefore implicitly assume only deadline type of timing constraints on tasks in this section, though other types of constraints (as explained in Sec. 1.7) may occur in practice. real-time tasks cab be classified into the following three broad categories:

A real-time task can be classified into either hard, soft, or firm real-time task depending on the consequences of a task missing its deadline.

It is not necessary that all tasks of a real-time application belong to the same category. It is possible that different tasks of a real-time system can belong to different categories. We now elaborate these three types of real-time tasks.

Hard Real-Time Tasks: A hard real-time task is one that is constrained to produce its results within certain predefined time bounds. The system is considered to have failed whenever any of its hard real-time tasks does not produce its required results before the specified time bound.

An example of a system having hard real-time tasks is a robot. The robot cyclically carries out a number of activities including communication with the host system, logging all completed activities, sensing the environment to detect any obstacles present, tracking the objects of interest, path planning, effecting next move, etc. Now consider that the robot suddenly encounters an obstacle. The robot must detect it and as soon as possible try to escape colliding with it. If it fails to respond to it quickly (i.e. the concerned tasks are not completed before the required time bound) then it would collide with the obstacle and the robot would be considered to have failed. Therefore detecting obstacles and reacting to it are hard real-time tasks.

Another application having hard real-time tasks is an anti-missile system. An anti-missile system consists of the following critical activities (tasks). An anti-missile system must first detect all incoming missiles, properly position the anti-missile gun, and then fire to destroy the incoming missile before the incoming missile can do any damage. All these tasks are hard real-time in nature and the anti-missile system would be considered to have failed, if any of its tasks fails to complete before the corresponding deadlines.

Applications having hard real-time tasks are typically safety-critical (Can you think an example of a hard real-time system that is not safety-critical? [1]) This means that any failure of a real-time task, including its failure to meet the associated deadlines, would result in severe consequences. This makes hard real-time tasks extremely critical. Criticality of a task can range from extremely critical to not so critical. Task criticality therefore is a different dimension than hard or soft characterization of a task. Criticality of a task is a measure of the cost of a failure — the higher the cost of failure, the more critical is the task.

For hard real-time tasks in practical systems, the time bounds usually range from several micro seconds to a few milli seconds. It may be noted that a hard real-time task does not need to be completed within the shortest time possible, but it is merely required that the task must complete within the specified time bound. In other words, there is no reward in completing a hard real-time task much ahead of its deadline. This is an important observation and this would take a central part in our discussions on task scheduling in the next two chapters.

Firm Real-Time Tasks: Every firm real-time task is associated with some predefined deadline before which it is required to produce its results. However, unlike a hard real-time task, even when a firm real-time task does not complete within its deadline, the system does not fail. The late results are merely discarded. In other words, the utility of the results computed by a firm real-time task becomes zero after the deadline. Fig. 14 schematically shows the utility of the results produced by a firm real-time task as a function of time. In Fig. 14 it can be seen that if the response time of a task exceeds the specified deadline, then the utility of the results becomes zero and the results are discarded.

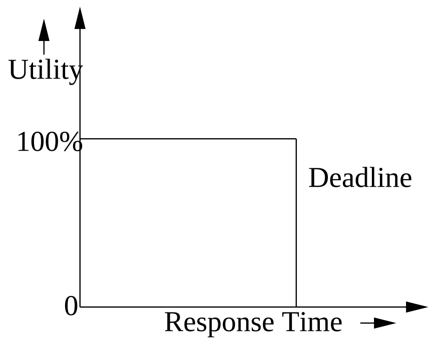

Figure 14: Utility of Result of a Firm Real-Time Task with Time

Firm real-time tasks typically abound in multimedia applications. The following are two examples of firm real-time tasks:

- **Video conferencing:** In a video conferencing application, video frames and the accompanying audio are converted into packets and transmitted to the receiver over a network. However, some frames may get delayed at different nodes during transit on a packet-switched network due to congestion at different nodes. This may

[1] Some computer games have hard real-time tasks, these are not safety-critical though. Whenever a timing constraint is not met, the game may fail, but the failure may at best be a mild irritant to the user.

result in varying queuing delays experienced by packets travelling along different routes. Even when packets traverse the same route, some packets can take much more time than the other packets due to the specific transmission strategy used at the nodes. (More discussions on this issue is given in Chapter 7.) When a certain frame is being played, if some preceding frame arrives at the receiver, then this frame is of no use and is discarded. Due to this reason, when a frame is delayed by more than say one second, it is simply discarded at the receiver-end without carrying out any processing on it.

- **Satellite-based tracking of enemy movements:** Consider a satellite that takes pictures of an enemy territory and beams it to a ground station computer frame by frame. The ground computer processes each frame to find the positional difference of different objects of interest with respect to their position in the previous frame to determine the movements of the enemy. When the ground computer is overloaded, a new image may be received even before an older image is taken up for processing. In this case, the older image is of not much use. Hence the older images may be discarded and the recently received image could be processed.

For firm real-time tasks, the associated time bounds typically range from a few milli seconds to several hundreds of milli seconds.

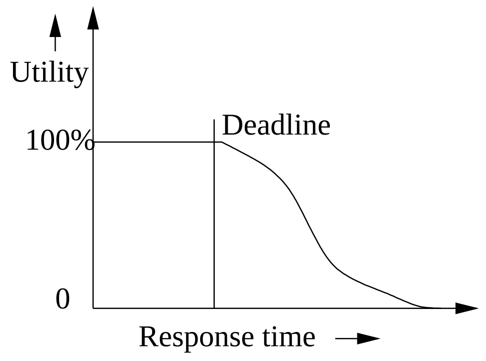

Figure 15: Utility of the Results Produced by a Soft Real-Time Task As A Function of Time

Soft Real-Time Tasks: Soft real-time tasks also have time bounds associated with them. However, unlike hard and firm real-time tasks, the timing constraints on soft real-time tasks are not expressed as absolute values. Instead, the constraints are expressed either in terms of the average response times required.

An example of a soft real-time task is web browsing. Normally, after an URL (**Uniform Resource Locater**) is clicked, the corresponding web page is fetched and displayed within a couple of seconds on the average. However, when it takes several minutes to display a requested page, we still do not consider the system to have failed, but merely express that the performance of the system has degraded.

Another example of a soft real-time task is a task handling a request for a seat reservation in a railway reservation application. Once a request for reservation is made, the response should occur within 20 seconds on the average. The response may either be in the form of a printed ticket or an apology message on account of unavailability of seats. Alternatively, we might state the constraint on the ticketing task as: At least in case of 95% of reservation requests, the ticket should be processed and printed in less than 20 seconds.

Let us now analyze the impact of the failure of a soft real-time task to meet its deadline, by taking the example of the railway reservation task. If the ticket is printed in about 20 seconds, we feel that the system is working fine and get a feel of having obtained instant results. As already stated, missed deadlines of soft real-time tasks do not result in system failures. However, the utility of the results produced by a soft real-time task falls continuously with time after the expiry of the deadline as shown in Fig. 15. In Fig. 15, the utility of the results produced are 100%

if produced before the deadline, and after the deadline is passed the utility of the results slowly falls off with time. For soft real-time tasks that typically occur in practical applications, the time bounds usually range from a fraction of a second to a few seconds.

Non-Real-Time Tasks: A non-real-time task is not associated with any time bounds. Can you think of any example of a non-real-time task? Most of the interactive computations you perform now a days are handled by soft real-time tasks. However, about two or three decades back, when computers were not interactive almost all tasks were non-real-time. A few examples of non-real-time tasks are: batch processing jobs, e-mail, and back ground tasks such as event loggers. You may however argue that even these tasks, in the strict sense of the term, do have certain time bounds. For example, an e-mail is expected to reach its destination at least within a couple of hours of being sent. Similar is the case with a batch processing job such as pay-slip printing. What then really is the difference between a non-real-time task and a soft real-time task? For non-real-time tasks, the associated time bounds are typically of the order of a few minutes, hours or even days. In contrast, the time bounds associated with soft real-time tasks are at most of the order of a few seconds.

7 Timing Constraints

We have seen that the correctness of real-time tasks depend both on the logical correctness of the result as well as on the satisfaction of the corresponding timing constraints. The timing constraints as we shall see in this Section, in fact apply to certain events in a system. These events may be generated by the tasks themselves or the environment of the system. An example of such an event is the event of activation of a motor. Remember that the results may be generated at different times and it may not be in the form of a single one time result. We must first properly characterize the events in a system, to understand the timing behavior of real-time systems.

7.1 Events in a Real-Time System

An event may be generated either by the system or its environment. Based on this consideration, events can be classified into the following two types:

Stimulus Events: Stimulus events are generated by the environment and act on the system. These events can be produced asynchronously (i.e. aperiodically). For example, a user pressing a button on a telephone set generates a stimulus event to act on the telephone system. Stimulus events can also be generated periodically. As an instance, consider the periodic sensing of the temperature of the reactor in a nuclear plant.

Response Events: Response events are usually produced by the system in response to some stimulus events. Response events act on the environment. For example, consider a chemical plant where as soon as the temperature exceeds 100°C, the system responds by switching off the heater. Here, the event of temperature exceeding 100°C is the stimulus and switching off of the heater is the response. Response events can either be periodic or aperiodic.

An event may either be instantaneous or may have certain duration. For example, a button press event is described by the duration for which the button was kept pressed. Some authors argue that durational events are really not a basic type of event, but can be expressed using other events. In fact, it is possible to consider a duration event as a combination of two events: a start event and an end event. For example, the button press event can be described by a combination of 'start button press' and 'end button press' events. However, it is often convenient to retain the notion of a durational event. In this text, we consider durational events as a special class of events. Using the preliminary notions about events discussed in this subsection, we classify various types of timing constraints in subsection 1.7.1.

7.2 Classification of Timing Constraints

A classification of the different types of timing constraints is important. Not only would it give us an insight into the different types of timing constraints that can exist in a system, but it can also help us to quickly identify the different timing constraints that can exist from a casual examination of a problem. That is, in addition to better understanding of the behavior of a system, it can also let us work out the specification of a real-time system accurately.

Different timing constraints associated with a real-time system can broadly be classified into performance and behavioral constraints.

Performance constraints are the constraints that are imposed on the response of the system. Behavioral constraints are the constraints that are imposed on the stimuli generated by the environment.

Behavioral constraints ensure that the environment of a system is well behaved, whereas performance constraints ensure that the computer system performs satisfactorily.

Each of *performance* and *behavioral* constraints can further be classified into the following three types:

- Delay Constraint
- Deadline Constraint
- Duration Constraint

We now explain these three classes of constraints:

Delay: A delay constraint captures the *minimum* time (delay) that must elapse between the occurrence of two arbitrary events e_1 and e_2. After e_1 occurs, if e_2 occurs earlier than the *minimum delay*, then a delay violation is said to occur. A delay constraint on the event e_2 can be expressed more formally as follows:

$$t(e_2) - t(e_1) \geq d$$

Where $t(e_2)$ and $t(e_1)$ are the time stamps on the events e_2 and e_1 respectively and d is the *minimum delay* specified from e_2. A delay constraint on the events e_2 with respect to the event e_1 is shown pictorially in Fig. 16. In Fig. 16, Δ denotes the actual separation in time between the occurrence of the two events e_1 and e_2 and d is the required minimum separation between the two events (delay). It is easy to see that e_2 must occur after at least d time units have elapsed since the occurrence of e_1, otherwise we shall have a delay violation.

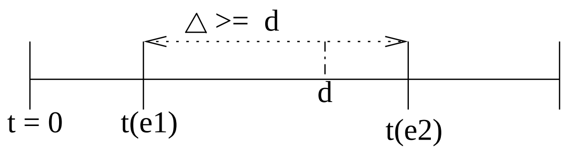

Figure 16: Delay Constraint Between Two Events e1 and e2

Deadline: A deadline constraint captures the permissible *maximum* separation between any two arbitrary events e_1 and e_2. In other words, the second event (i.e. e_2) must follow the first event (i.e. e_1) within the permissible maximum separation time. Consider that $t(e_1)$ and $t(e_2)$ are the time stamps on the occurrence of the events e_1 and e_2 respectively and d is the *deadline* as shown in Fig. 17. In Fig. 17, Δ denotes the actual separation between the time of occurrence of the two events e_1 and e_2, and d is the deadline. A deadline constraint implies that e_2

must occur within d time units of e_1's occurrence. We can alternatively state that $t(e_1)$ and $t(e_2)$ must satisfy the constraint:

$$t(e_2) - t(e_1) \leq d$$

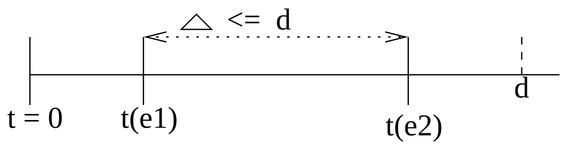

Figure 17: Deadline Constraint Between Two Events e1 and e2

The deadline and delay constraints can further be classified into two types each based on whether the constraint is imposed on the stimulus or on the response event. This has been explained with some examples in Sec. 7.2.

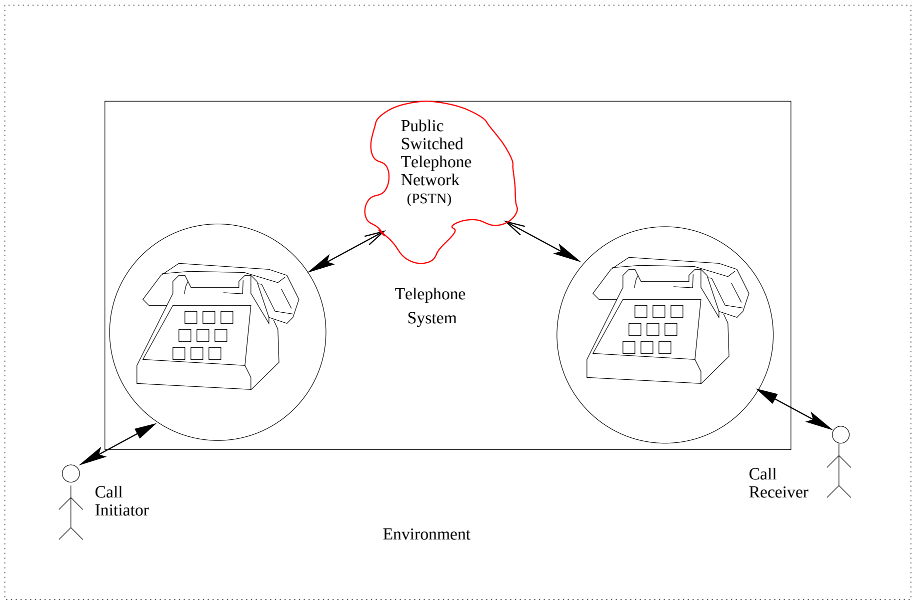

Figure 18: Schematic Representation of a Telephone System

Duration: A duration constraint on an event specifies the time period over which the event acts. A duration constraint can either be minimum type or maximum type. The minimum type duration constraint requires that once the event starts, the event must not end before a certain minimum duration; whereas a maximum type duration constraint requires that once the event starts, the event must end before a certain maximum duration elapses.

7.3 Examples of Different Types of Timing Constraints

We illustrate the different classes of timing constraints by using the examples from a telephone system discussed in [1]. A schematic diagram of a telephone system is given in Fig. 18. Note that I have intentionally drawn an old styled telephone, because its operation is easier to understand! Here, the telephone handset and the Public Switched Telephone Network (PSTN) are considered as constituting the computer system and the users as forming the environment. In the following, we give a few simple example operations of the telephone system to illustrate the different types of timing constraints.

Deadline constraints: In the following, we discuss four different types of deadline constraints that may be identified in a real-time system depending on whether the two events involved in a deadline constraint are stimulus type or response type.

Stimulus–Stimulus(SS): In this case, the deadline is defined between two stimuli. This is a behavioral constraint, since the constraint is imposed on the second event which is a stimulus. An example of an SS type of deadline constraint is the following:
`Once a user completes dialling a digit, he must dial the next digit within the next 5 seconds. Otherwise an idle tone is produced.`

In this example, the dialing two consecutive digits represents the two stimuli to the telephone system.

Stimulus–Response(SR): In this case, the deadline is defined on the response event, measured from the occurrence of the corresponding stimulus event. This is a performance constraint, since the constraint is imposed on a response event. An example of an SR type of deadline constraint is the following:
`Once the receiver of the hand set is lifted, the dial tone must be produced by the system within 2 seconds, otherwise a beeping sound is produced until the handset is replaced.`

In this example, the lifting of the receiver hand set represents a stimulus to the telephone system and production of the dial tone is the response.

Response–Stimulus(RS): Here the deadline is on the production of response counted from the corresponding stimulus. This is a behavioral constraint, since the constraint is imposed on the stimulus event. An example of an RS type of deadline constraint is the following:
`Once the dial tone appears, the first digit must be dialed within 30 seconds, otherwise the system enters an idle state and an idle tone is produced.`

Response–Response(RR): An RR type of deadline constraint is defined on two response events. In this case, once the first response event occurs, the second response event must occur before a certain deadline. This is a performance constraint, since the timing constraint has been defined on a response event. An example of an RR type of deadline constraint is the following:
`Once the ring tone is given to the callee, the corresponding ring back tone must be given to the caller within two seconds, otherwise the call is terminated.`

Here ring back tone and the corresponding ring tone are the two response events.

Delay Constraint:
We can identify only one type of delay constraint (SS type) in the telephone system example that we are considering. However, in other problems it may be possible to identify different types of delay constraints. An SS type of a delay constraint is a behaviorial constraint. An example of an SS type of delay constraint is the following:
`Once a digit is dialled, the next digit should be dialled after at least 1 second. Otherwise, a beeping sound is produced until the call initiator replaces the handset.`

Here the delay constraint is defined on the event of dialling of the next digit (stimulus) after a digit is dialled (also a stimulus).

Duration:
A duration constraint on an event specifies the time interval over which the event acts. An example of a duration constraint is the following:
`If you press the button of the handset for less than 15 seconds, it connects to the local operator. If you press the button for any duration lasting between 15 to 30 seconds, it connects to the international operator. If you keep the button pressed for more than 30 seconds, then on releasing it would produce the dial tone.`

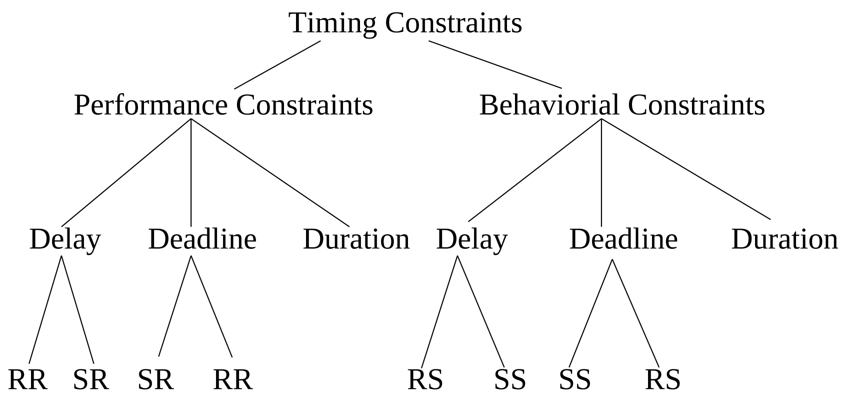

Figure 19: Classification of Timing Constraints

A classification of the different types of timing constraints that we discussed in this section is shown in Fig. 19. Note that a performance constraint can either be delay, deadline, or durational type. The delay or deadline constraints on performance can either be RR or RS type. Similarly, the behaviorial constraints can either be delay, deadline, or durational type. The delay or deadline constraints on behavior of environment can either be RS or SS type.

8 Modelling Timing Constraints

In this section we describe how the timing constraints identified in Sec. 1.7.3 can be modelled. Modelling time constraints is very important since once a model of the time constraints in a system is constructed, it can serve as a formal specification of the system. Further, if all the timing constraints in a system are modelled accurately, then it may even be used to automatically generate code from it. Besides serving as a specification, modelling time constraints can help to verify and understand a real-time system.

The modelling approach we discuss here is based on *Finite State Machines* (FSMs). An FSM is a powerful tool which has long been used to model traditional systems. In an FSM, a state is defined in terms of the values assumed by some attributes. For example, the states of an elevator may be denoted in terms of its directions of motion. Here direction is the attribute, based on which the states up, down, and stationery are defined.

In an FSM model, at any point of time a system can be in any one of a (possibly infinite) number of states. A state is represented by a circle. The system changes state due to events that change the values of, or relations among the state variables. A state change is also called a state transition. A transition causing event may either be an interface event that are transmitted between the environment and the computer system or it could also be an internal event that is generated and consumed solely within the system. A transition from one state to another is represented by drawing a directed arc from the source to the destination (see Fig. 20). The event causing a transition is annotated on the arc. We keep our discussions of FSM to the bare minimum since we assume that the reader is familiar with basic FSM modelling of traditional systems.

We use an *Extended Finite State Machine* (EFSM) to model time constraints. EFSM extends the traditional FSM by incorporating the action of setting a timer and the expiry event of a timer. The notations we use for construction of EFSMs are simple and straightforward. Therefore rather than introducing them formally, we have illustrated them through an example in Fig. 20. The example shown in Fig. 20 describes that if an event e1 occurs when the current state of the system is s1, then an action will be taken by setting a timer to expire in the next 20 milli seconds and the system transits to state s2.

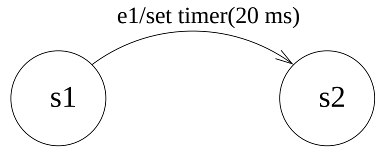

Figure 20: Conventions Used in Drawing an EFSM

We have already discussed that events can be considered to be of two types: stimulus events and response events. We had also discussed different types of timing constraints in Section 1.7.3. Now we explain how these constraints can be modelled by using EFSMs.

Stimulus-Stimulus (SS) constraints: Let us consider the example of an SS type of deadline constraint we had discussed in Section 1.7.3: `Once the first digit has been dialled on the telephone handset, the next digit must be dialled within the next 5 milli seconds.` This has been modelled in Fig. 21. In Fig. 21 observe that as soon as the first digit is dialled, the system enters the "Await Second Digit" state and the timer is set to 20 milli seconds. If the next digit does not appear within 20 milli seconds, then the timer alarm expires and the system enters the "Await Caller On-hook" state and a beeping sound is produced. If the second digit occurs before 20 milli seconds, then the system transits to the "Await Next Digit" state.

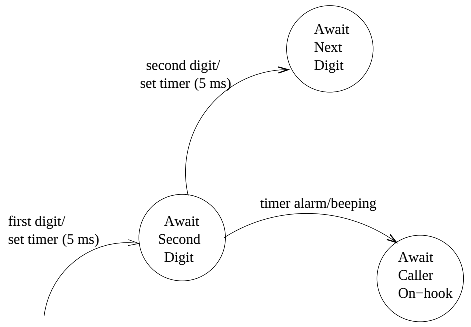

Figure 21: Model of an SS Type of Deadline Constraint

Response–Stimulus(RS) constraint: In Sec. 1.7.3, we had considered the following example of an RS type of deadline constraint: `Once the dial tone appears, the first digit must be dialed within 30 seconds, otherwise the system enters an idle state and an idle tone is produced.`

The EFSM model for this constraint is shown in Fig. 22. In Fig. 22, as soon as dial tone appears, a timer is set to expire in 30 seconds and the system transits to the "Await First Digit" state. If the timer expires before the first digit arrives, then the system transits to an idle state where an idle tone is produced. Otherwise, if the digit appears first, then the system transits to the "Await Second Digit" state.

Stimulus–Response(SR): In Sec. 1.7.3, we had considered the following example of an SR type of deadline constraint: `Once the receiver of the hand set is lifted, the dial tone must be produced by the system within 20 seconds, otherwise a beeping sound is produced until the handset is replaced.`

The EFSM model for this constraint is shown in Fig. 23. As soon as the handset is lifted, a timer is set to expire after 2 sec and the system transits to "Await Dial Tone" state. If the dial tone appears first, then the system transits

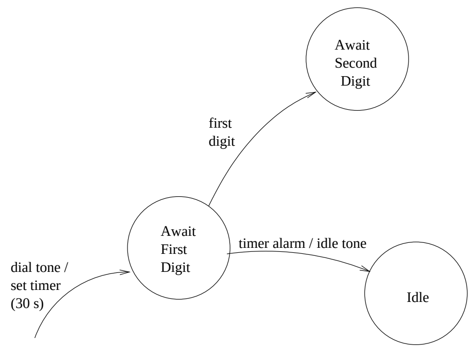

Figure 22: Model of an RS Type of Deadline Constraint

to "Await First Digit" state. Otherwise, it transits to "Await Receiver On-hook" state.

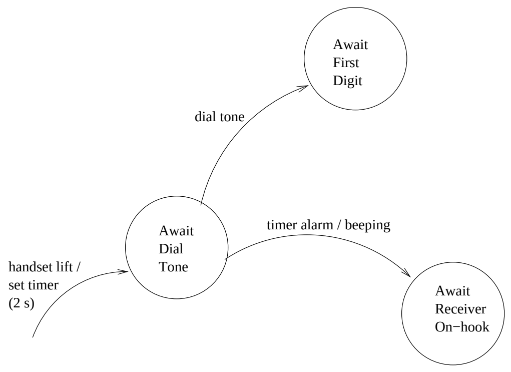

Figure 23: Model of an SR Type of Deadline Constraint

Response–Response(RR): In Sec. 1.7.3, we had considered the following example of an RR type of constraint: `Once the ring tone is given to the callee, the corresponding ring back tone must be given to the caller within two seconds, otherwise the call is terminated.`

The EFSM model for this constraint is shown in Fig. 24. In Fig. 24, as soon as the ring tone is produced, the system transits to await ring-back tone state and a timer is set to expire in 2 Secs. If the ring-back tone appears first, the system transits to "Await First Digit" state, else it enters "Await Receiver On-hook" state, and the call is terminated.

Delay Constraint: A delay constraint between two events is one where after an event occurs, a minimum time must elapse before the other event can occur. We had considered the following example of delay constraint in Sec. 1.7.3: `After a digit is dialed, the next digit should be dialed no sooner than 10 milli seconds.` The EFSM model for it is shown in Fig. 25. In Fig. 25, if the next digit appears before the alarm, then the beeping sound is produced and the system transits to "await caller on-hook" state.

Durational Constraint: In case of a durational constraint, an event is required to occur for a specific duration. The example of a durational constraint we had considered in Sec. 1.7.3 is the following: `If you press the button of the handset for less than 15 seconds it connects to the local operator. If you press the button for any duration lasting between 15 to 30 seconds, it connects to the international operator. If you keep the button pressed for more than 30 seconds, then on releasing it would produce the dial`

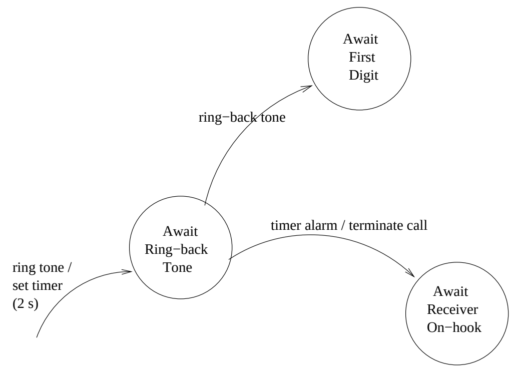

Figure 24: Model of an RR Type of Deadline Constraint

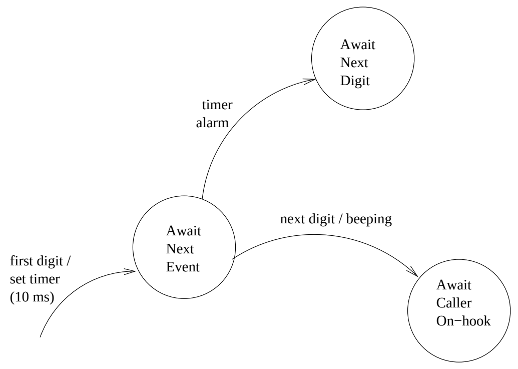

Figure 25: Model of an SS Type of Delay Constraint

`tone.`

The EFSM model for this example is shown in Fig. 26. Note that we have introduced two intermediate states "Await Event " and "Await Event 2" to model a durational constraint.

SUMMARY

- The main aim of this Chapter was to introduce you to some very basic concepts and terminologies associated with real-time systems that would be used throughout this book.
- A system is said to be *real-time* when quantitative expressions of time are necessary to describe the behavior of the system.
- A real-time task is one that is associated with some time constraints.
- A real-time task is classified into either hard, firm, or soft real-time type depending on the consequences of a task failing to meet its timing constraints.
- A safety-critical system is one which does not have a fail-safe state and any failure of the system can cause severe damage. Many hard real-time systems are safety-critical in nature.
- The typical characteristic features of a hard real-time system include embedded, feedback and distributed structure, and safety-criticality. It is possible though that some hard real-time systems may not have these features.

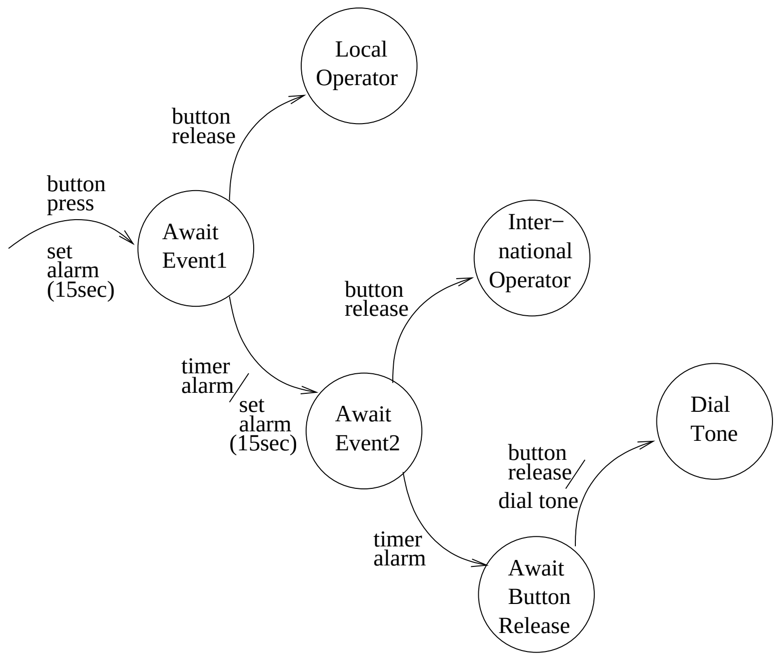

Figure 26: A Model of a Durational Constraint

- There are two main categories of time constraints in embedded hard real-time systems: performance constraints and behavioral constraints. Performance constraints are timing constraints imposed on the response of the controlling computer system, whereas behaviorial constraints are timing constraints imposed on the behavior of the environment. The performance constraints may not be guaranteed if the behaviorial constraints are not satisfied.
- We discussed how to model the performance and behaviorial types of constraints using an Extended Finite State Machine (EFSM) that extends the traditional FSM with the concepts of setting a timer and the corresponding generation of a timer alarm event. Such a model in addition to serve as a specification technique, can also help in verufucation and automatic code generation.

EXERCISES

1. State whether you consider the following statements to be **TRUE** or **FALSE**. Justify your answer in each case.
   (a) A hard real-time application is made up of only hard real-time tasks.
   (b) Every safety-critical real-time system has a fail-safe state.
   (c) A deadline constraint between two stimuli can be considered to be a behaviorial constraint on the environment of the system.
   (d) Hardware fault-tolerance techniques can easily be adapted to provide software fault-tolerance.
   (e) A good algorithm for scheduling hard real-time tasks must try to complete each task in the shortest time possible.
   (f) All hard real-time systems are safety-critical in nature.

(g) Performance constraints on a real-time system ensure that the environment of the system is well-behaved.

(h) Soft real-time tasks are those which do not have any time bounds associated with them.

(i) Minimization of average task response times is the objective of any good hard real-time task-scheduling algorithm.

(j) It should be the goal of any good real-time operating system to complete every hard real-time task as ahead of its deadline as possible.

2. What do you understand by the term "real-time"? How is the concept of real-time different from the traditional notion of time? Explain your answer using a suitable example.

3. What does the term "real" in a real-time system signify? Explain what do you mean by a real-time system.

4. Using a block diagram show the important hardware components of a real-time system and their interactions. Explain the roles of the different components.

5. In a real-time system, raw sensor signals need to be preprocessed before they can be used by a computer. Why is it necessary to preprocess the raw sensor signals before they can be used by a computer? Explain the different types of preprocessing that are normally carried out on sensor signals to make them suitable to be used directly by a computer.

6. Identify the key differences between hard real-time, soft real-time, and firm real-time systems. Give at least one example of real-time tasks corresponding to these three categories. Identify the timing constraints in your tasks and justify why the tasks should be categorized into the categories you have indicated.

7. Explain the key differences between the characteristics of a soft real-time task such as web browsing and a non-real-time task such as e-mail delivery.

8. Give an example of a soft real-time task and a non-real-time task. Explain the key difference between the characteristics of these two types of tasks.

9. Name any two important sensor devices and two actuator devices used in real-time applications and explain the physical principles behind their working.

10. Draw a schematic model showing the important components of a typical hard real-time system. Explain the working of the output interface using a suitable schematic digram. Explain using a suitable circuit diagram how digital to analog (DAC) conversion can be achieved in an output interface.

11. Draw a schematic model showing the important components of a typical hard real-time system. Explain the working of the input interface using a suitable schematic digram. Explain using a suitable circuit diagram how analog to digital (ADC) conversion is achieved in an input interface.

12. In a hard real-time system, is it necessary that every task in the system be of hard real-time type? Explain your answer using a suitable example.

13. In the context of providing system-level fault-tolerance, why are hardware failures more predictable and easier to handle compared to software failures? Explain the N-version scheme to provide software fault-tolerance. What are the shortcomings of this scheme? How does the recovery block technique attempt to overcome it? Does it succeed? Explain your answer.

14. Explain the checkpointing and rollback recovery scheme to provide fault-tolerant real-time computing. Explain the types of faults it can help tolerate and the faults it can not tolerate. Explain the situations in which this technique is useful.

15. Answer the following questions concerning fault-tolerance of real-time systems.

(a) Explain why hardware fault-tolerance is easier to achieve compared to software fault-tolerance.

(b) Explain the main techniques available to achieve hardware fault tolerance.

(c) What are the main techniques available to achieve software fault-tolerance? What are the shortcomings of these techniques?

16. Briefly explain how hardware failures (e.g. processor failures) can be tolerated in safety-critical hard real-time applications. Consider the fact that hard real-time tasks have stringent deadlines which must be met under all circumstances.

17. What do you understand by the "fail-safe" state of a system? Safety-critical real-time systems do not have a fail-safe state. What is the implication of this?

18. Explain why safety and reliability are not independent issues in safety-critical hard real-time systems. Explain the basic techniques you would adopt to develop a software product that is required to be highly reliable.

19. Is it possible to have an extremely safe but unreliable system? If your answer is affirmative, then give an example of such a system. If you answer in the negative, then justify why it is not possible for such a system to exist.

20. What is a safety-critical system? Give a few practical examples safety-critical hard real-time systems. Are all hard real-time systems safety-critical? If not, give at least one example of a hard real-time system that is not safety-critical.

21. Explain with the help of a schematic diagram how the recovery block scheme can be used to achieve fault-tolerance of real-time tasks. What are the shortcomings of this scheme? Explain situations where it can be satisfactorily be used and situations where it can not be used.

22. Identify and represent the timing constraints in the following air-defense system by means of an extended state machine diagram. Classify each constraint into either performance or behaviorial constraint.
Every incoming missile must be detected within 0.2 Seconds of its entering the radar coverage area. The intercept missile should be engaged within 5 Seconds of detection of the target missile. The intercept missile should be fired after 0.1 Seconds of its engagement but no later than 1 Sec.

23. Represent a wash-machine having the following specification by means of an extended state machine diagram. The wash-machine waits for the `start` switch to be pressed. After the user presses the `start` switch, the machine fills the wash tub with either hot or cold water depending upon the setting of the `HotWash` switch. The water filling continues until the high level is sensed. The machine starts the agitation motor and continues agitating the wash tub until either the preset timer expires or the user presses the `stop` switch. After the agitation stops, the machine waits for the user to press the `start Drying` switch. After the user presses the `startDrying` switch, the machine starts the hot air blower and continues blowing hot air into the drying chamber until either the user presses the `stop` switch or the preset timer expires.

24. In a real-time system what is the difference between a performance constraint and a behaviorial constraint? Give practical examples of each type of constraint.

25. Represent the timing constraints in a collision avoidance task in an air surveillance system as an extended finite state machine (EFSM) diagram. The collision avoidance task consists of the following activities.

    - The first subtask named radar signal processor processes the radar signal on a signal processor to generate the track record in terms of the target's location and velocity within 100 mSec of receipt of the signal.
    - The track record is transmitted to the data processor within 1 mSec after the track record is determined.
    - A subtask on the data processor correlates the received track record with the track records of other targets that come close to detect potential collision that might occur within the next 500 mSec.
    - If a collision is anticipated, then the corrective action is determined within 10 mSec by another subtask running on the data processor.

- The corrective action is transmitted to the track correction task within 25 mSec.

26. Consider the following (partial) specification of a real-time system:
The velocity of a space-craft must be sampled by a computer on-board the space-craft at least once every second (the sampling event is denoted by S). After sampling the velocity, the current position is computed (denoted by event C) within 100msec, parallelly the expected position of the space-craft is retrieved from the database within 200msec (denoted by event R). Using these data, the deviation from the normal course of the space-craft must be determined within 100 msec (denoted by event D) and corrective velocity adjustments must be carried out before a new velocity value is sampled in (the velocity adjustment event is denoted by A). Calculated positions must be transmitted to the earth station at least once every minute (position transmission event is denoted by the event T).

    Identify the different timing constraints in the system. Classify these into either performance or behaviorial constraints. Construct an EFSM to model the system.

27. Construct the EFSM model of a telephone system whose (partial) behavior is described below:

    After lifting the receiver handset, the dial tone should appear within 20 seconds. If a dial tone can not be given within 20 seconds, then an idle tone is produced. After the dial tone appears, the first digit should to be dialled within 10 seconds and the subsequent five digits within 5 seconds of each other. If the dialling of any of the digits is delayed, then an idle tone is produced. The idle tone continues until the receiver handset is replaced.

28. What are the different types of timing constraints that can occur in a system. Give examples of each.

References

[1] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods for validating them. *IEEE Transactions on Software Engineering*, Vol. 11(No. 1):80–86, January 1985.

[2] Steve Heath. *Embedded Systems Design.* Elsevier, 2003.

[3] Brian Santo. Embedded battle royale. *IEEE Spectrum*, pages 36–42, December 2001.